Preface

This book is organised into a large number of brief, self-contained entries.

Admittedly, there is no such thing as a *self-contained entry*. For example, you need some knowledge of English to understand this paragraph. But, the principle is that each entry, of one or two pages, is a conceptual whole as well as a part of a greater whole (see note 20) in the same way that a car has four whole wheels, and not eight half wheels.

Some entries are intended to demonstrate a technique, or introduce an historically contingent fact such as the actual syntax of a contemporary language, or in this case, a specific issue regarding this book. Others are intended to illustrate a more eternal truth. They may be about a contemporary language, but stress a philosophical position or broadly based attitude. Both of these I have called *notions*. Finally, there are entries that are intended to cause the reader to do something other than just nodding their head as a sign of either agreement or an incipient dormant state. These are the exercises.

The distinction can only be arbitrary; the classification is merely a guide to suggest the sense in which the pages are intended.

In many cases, entries that are not specifically labelled as exercises involve generic opportunities for self-study. As this is a book on computer programming, it is natural and strongly advised that the reader try implementing each concept of interest as it arises. With this in mind, I have tried hard not to leave out pragmatic details whose omission would leave the reader with nothing but the illusion of understanding. Nevertheless, actually cutting practice code makes a big difference in the ability of the programmer to use the concepts when the need arises.

At the end of the book are the notes explaining short and simple issues or (paradoxically) issues that are too complex to explain in this book. If a note became too lengthy while being written it was converted into a notion or an exercise.

Chapter 2

A Grab Bag of Computational Models

In which we take the view that designing software is the technological aspect of computer science in analogy to the designing of hardware being the technological side of electronic science. We find that there is a smooth shift from one to the other, with firmware in the twilight zone.

Knowing that a hardware engineer or technician requires a grab bag full of formal models of the material at hand, small enough and simple enough to submit to analysis, realistic enough to be relevant, we admit that a programmer likewise needs a collection of software models: pure archetypical computational mechanisms that assist analysis and design of practical software in the real and very impure world.

We recognise that every piece of software is a virtual machine. And so, study a collection of specific abstract models, including Turing machines, state machines, Von Neumann machines, s-code reduction, lambda calculus, primitive recursive functions, pure string substitution expression reduction, etc.

We learn about unification-reduction, which has been rightly referred to as the arithmetic of computer science, acting both as a low- and high-level concept. It is a first model of every computer language so far devised. The substitution of equals for equals is a beguilingly simple concept; we learn that it is a deeply powerful representation of computation itself. Computation is constructive logic, the propositional and predicate calculi being the foundational material.

Notion 8: Abstract and Virtual Machines

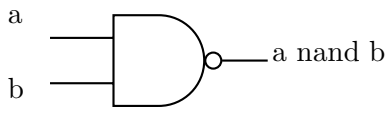
We do not know how the universe actually works.

Through whatever process pleases us, scientific or otherwise, we decide to act on incompletely justified assumptions about the possible effects of our actions. A physical machine, be it a can-opener or a computer, is always designed as an idealised conception in our minds. For digital computers, we construct small component machines whose behaviour may be finitely described.

The nand gate takes two inputs, whose values may only be 0 or 1, and so the output can be listed explicitly for each of the four possible input combinations.

a	b	a `nand` b
0	0	1
0	1	1
1	0	1
1	1	0

We may also conceptualise a component as a physical device.



While this might (or might not depending on your background) appeal more strongly to your intuition, it is still a virtual machine, an abstract construction, or an idealised conception of our minds.

Inspired by this concept, we build a physical device that is supposed to work in like manner. In reality it never does. If we are smart then we know that it does not. But it works correctly, under the right conditions, to sufficient accuracy, with sufficient probability, to make it practical to assume that it will work.

Abstract devices abound. They include everything from idealised can-openers to spacecraft complete with navigational software and zero-

gravity toilet. But, in this discussion we concentrate on abstract *digital computational* machines. Generically this will involve a finite symbolic state that changes in time.

Any computer language defines an abstract machine.

The distinction between an abstract machine and a virtual machine is that we have an implementation of the virtual machine.

Pragmatically, there is little to distinguish the nature of the firmware or hardware virtual machine from the pure software virtual machine.

For example, Java is said to operate on the Java Virtual Machine or JVM, which is typically implemented in software. But we could equally build a JVM chip. The JVM is just an orthodox Von Neumann architecture, and would be easy to design and manufacture. We could also build a CVM to run our C programs. In a strong sense that is exactly what the compiler is.

Micro-coded machines have machine code in which each instruction is actually a small program written in a simpler lower-level machine code. Thus, the supposed hardware, the target for an assembler, is actually being emulated on even lower level-hardware.

Software is easy to modify; firmware can be modified with moderate effort; and hardware is typically difficult to change. As components become smaller towards the size of atoms and electrons, we find less ability to control them directly via high-level software in our machine, but there is no precise cutoff.

The difficulty of modifying hardware is contingent. Programmable gate arrays can be modified in normal operation. Research is ongoing into ways in which the arrangement of transistors on the chip may be dynamically modified. In the longer run, hardware may be just another form of software.

Conceptually, they are all virtual machines.

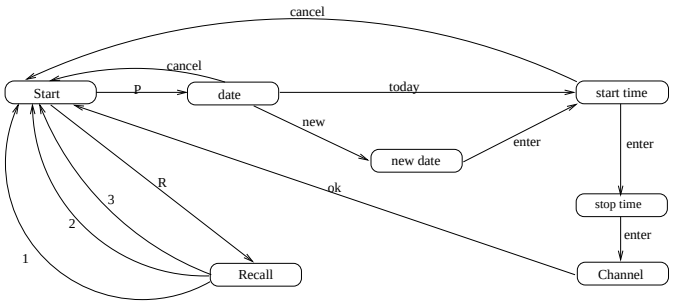
Notion 9: State Machines

The interactive state machine concept is central computational technology. To apply this concept we first identify its four components. The device must be distinct from its environment. A wristwatch, for example, is distinct from its wearer. The device must internalise information. A wristwatch stores the time. The device must act externally. A wristwatch may display the time, or sound an alarm. The user must act on the device. The wearer may push buttons or turn knobs on the watch. Finally, the device must exist in time, responding to actions of the environment by actions of its own, and by modifying its stored information. Any computer, analogue or digital, is a state machine.

Discreteness is definitive of digital technology. Quantities are discrete if each may be distinguished by a definite amount from all others. The display of a digital watch is discrete because the numerals are distinct from each other. In contrast, the possible positions of the second hand of an analogue watch form a continuum. No matter how good our eyesight, there are always two locations so close together that we cannot tell them apart. Pushing a button is a discrete action — we either manage to push the button, or we do not. Turning a knob is continuous; we may turn the knob a little or a lot, with indefinite shades in between. A digital watch might act only ten times a second, it operates at discrete times. An analogue watch responds continuously to the continuous turn of the knob. The state of the analogue watch is a continuous voltage or position, while the digital watch stores only a collection of discrete symbols.

On closer examination, most analogue watches are discrete state. The second hand moves by distinct jumps. But looked at even more closely, the jumps of the second hand are fast, but continuous. So are the changes in the display of a digital watch. It is an open question whether the universe is ultimately discrete or continuous. In practice, the question is resolved by asking which model most simply describes the interesting behaviour to the desired accuracy. In programming, we model a digital computer as having a discrete state, display, input and action. This is referred to as a discrete state machine. If the states, display, input, and actions performed in a finite time are all finite, we refer to this as a finite state-machine.

Another example of a discrete-state machine is a video cassette player interface. Each step is discrete. As we operate the machine we switch it from one state to another. The validity of an input varies from state to state. The video player responds by showing information on its display and updating the information that it stores.



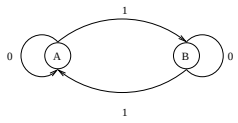
As above, we can draw a diagram, a network of nodes and links that represents the states and transitions of the state machine. The input is written on the links, and the output (not indicated in the above diagram) may be thought of as being dependent on the state. This notion may be given a much more precise formalism. (see page 103).

A state machine may be used to map input strings to output strings (see page 22). This can either be used as a program, mapping an input question to the output response, or as a temporal interaction for interactive systems such as communication protocol implementation, and user interface design.

Explicit use of state machines is most important for embedded controllers and communication devices. Often such a machine is written up as an array of transition and state information (see page 110). State machines can be implemented easily by a microprocessor, but also rather nicely by regular arrays of logic gates.

Notion 10: State Machines in Action

State machines (see page 20) can operate on symbolic strings in a variety of ways. Fundamentally, it maps a symbolic string to a sequence of states and transitions. If we associate a symbol with the state (a Moore machine) or transition (a Mealy machine), then we have a string-to-string map. Alternatively, by taking the last symbol of the output string, we obtain a string to symbol map in either case.



The state machine on the left responds to a {0,1} string on its input with an {A,B} string on its output. If the machine starts in state A, then 0011010 is mapped to AAABAABB. The machine will be in state B exactly when the number of 1s so far is odd. Thus, state B is the odd parity state, and state A is the even parity state. The final symbol (in this case B) shows us the parity of the whole input string.

Indicating state A as the starting state, this machine is said to recognise, at state B, even parity strings. The final state, A or B, classifies the input string as even or odd parity.

For a Moore machine the output string is one symbol longer than the input string. For a Mealy machine the lengths are the same. Sometimes, since the starting symbol does not depend on the input string, the starting symbol is ignored in a Moore machine. But in the above example the parity of an empty string is even, and the output `"A"` is correct. A Mealy machine would not provide this output, but we can fix this by providing a *start of string* indicator. By using these and similar tricks, either machine can be used equally.

In the parity example, there is an output symbol for every state. We may relax this condition so that some states, or transitions, do not generate an output symbol. In this case, the string mapping behaviour of the Moore and Mealy machines is identical.

Explicit coding of state machines is most typically advisable in temporal interaction. User interfaces, communications systems, parsing of languages, and embedded control code all can benefit from this approach.

A state machine may be hard-coded by a systematic use of nested conditionals, with a variable storing a state number. The state number is tested and set to the new state. Procedural code and state machine code tend to fight each other. It is still possible for them to coexist, even call each other. But they should normally be written separately, and from a different point of view.

```
state = 0;
while(state<2)
{
  putchar(state==0?'A':'B'));
  c = getchar();
  if(state==0 && c=='0') state=0; else
  if(state==0 && c=='1') state=1; else
  if(state==1 && c=='0') state=1; else
  if(state==1 && c=='1') state=0; else
  state=2;
}
putchar(state==0?'A':'B'))
```

Rather than hard-coding the machine, an array can store output for given state. Another array can store new state versus old state and input. Generic code can then be used for the heart of the machine.

```
state = 0;
while(putchar(output[state]))
{
input = parse(getchar());
if(input==error) break;
state = next[state,input];
}
```

In some cases, it is best for the arrays to be replaced by functions, since either the size of the state machine or a lack of knowledge of its structure may prevent explicit listing of the states and state transitions.

Typically, the table driven approach (see page 110) to implementing a state machine is the most practical, as well as being the closest to the formal algebra (see page 103). This close association between the most formal and the most pragmatic approach to a datatype is very common, more than is often realised, and should be design focus.

Exercise 1: Virtual Machines

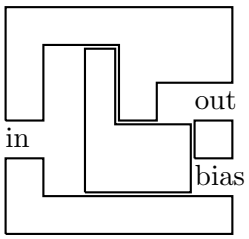
Design a pneumatic digital computer. (See hints below.)

Separate in your mind computers from electronics. The first fully fledged digital computer designed (by Charles Babbage) was mechanical, and was largely the same as the modern electronic digital computer. Pneumatics has many advantages over mechanics, e.g., an air hose can be bent around easily, while rods and wheels need careful alignment. Pneumatic computers are less affected by the environment and were used for industrial control into the 1980s.

The basic element in many computers is the nand-gate. It has two signal inputs and one output. If both inputs are active, then the output is inactive, otherwise the output is active. It computes "not both".

The simplest place to start is to design an inverter. It has one hose connector for input and one for output. If there is high pressure on the input then there is low pressure on the output; low pressure on the input means high pressure on the output.

In principle, an inverter can use an input to slide a block to shut off a high-pressure bias intake. If the input is low-pressure the bias escapes to the output, otherwise the output is low pressure. The one on the right is not practical because of difficulties such as sealing the sliding surfaces. For simplicity we can ignore this, but your solution is better if you consider the mechanics.



Two pressures can be used for each signal. A hi-lo combination means a logic 1, and a lo-hi means 0. An inverter might just swap the hoses. This is simple, but loses signal strength. The output should mainly be driven by a separate bias intake, which you can assume to be provided globally. Springs can also be used, from which a valve may be built.

Experiment and use your imagination.

Exercise 2: Finite State-Machines

A finite state-machine (see page 20) classifies strings (see page 22). Design finite state-machines that classifies strings according to —

1. whether it ends in the substring 1010, or not;
2. whether it contains the substring 1010, or not;
3. whether it is a binary number greater than 1010, or not;
4. its remainder after division by 4. (so, four categories).

In a standard msb first binary numeral, $x_3x_2x_1x_0$, x_3 is the most significant bit. The reverse order is lsb first, and is more natural for finite state-machines. To store two binary numbers we can use an infix $x_3x_2x_1x_0 + y_3y_2y_1y_0$, or spliced $x_3y_3x_1y_2x_1y_1x_0y_0+$, format.

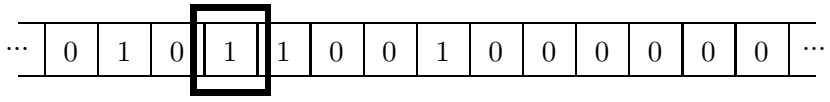
A finite state-machine (see page 20) maps strings (see page 22). Design state machines that map strings to the effect —

1. of incrementing an lsb first binary number;
2. of adding two spliced lsb first binary numbers;
3. of multiplying an lsb first number by 3;
4. of adding two msb first binary numbers;
5. of adding two lsb first infix binary numbers;
6. of multiplying two lsb first spliced binary numbers.

Some of the above operations are impossible for a finite state-machine — which ones and why?

Notion 11: Turing Machine

As a physical intuition, the definitive Turing machine is built from an infinite tape of discrete memory cells, together with an interactive cpu that moves along the tape reading and writing the cells. The cpu moves at a finite speed and has a finite number of states. Each cell contains one (at a time) of a finite collection of symbols. The choice of action, writing a symbol, moving to the left or right, or halting, is determined by a lookup table from the current tape symbol and cpu state.



The classic view is that each Turing machine computes a natural number function. The n symbols may be taken as the digits of a base-n natural number. At startup, all cells to the left of the cpu are 0, and only a finite number to the right are non-0. The initial tape represents a natural number. If the machine halts after a finite time, the tape will still only have a finite number of non-0 cells. The input number has been mapped to the output number. Generically, a Turing machine defines only a partial map, since the output is undefined if the machine never halts.

By finitely placing non-0 symbols to the left, we can program a Turing machine. For each program, the machine computes a potentially distinct function. But it is a generic limitation of effective computational devices that not all natural functions can be thus computed.

Other encodings can allow a Turing machine to solve problems on other domains. For example, a universal Turing machine takes a pair (m,d) of a Turing machine table m, and input data d, and emulates the machine operating on the data. Oddly, universality is subjective. Marvin Minksky's classic 7-state universal Turing machine works, but the

"proof" is mainly a subjective justification of the encoding.

To illustrate this point, consider the machine M that increments each even number, and deliberately does not halt on odd numbers. For each input (m,d) there is a unique output r. If the machine m does not halt on the data d, then we want the universal machine to not halt. If m halts on d, then encode (m,d) as an even number and encode r as the next odd number. Otherwise encode (m,d) as an odd number. Thus, M is universal. True, from any standard description of a Turing machine it is a non-computable problem to determine what to feed this universal machine. But logically, the representation is valid.

There are many Turing variations. Some have a lesser computational ability, but none have more. A one-way (single-pass) Turing machine moves only in one direction. Increment in binary can be performed by one pass, but this is so strong a restriction that this variants ability is reduced.

There are multi-tape and multi-cpu versions, as well as a two-dimensional one that can move north, east, south, and west. Trying to envisage this physically can lead to tangles of tape all over the floor. For simplicity, assume that the tapes and cpu's pass through each other.

The tapes may be fed around in circles, forming limited storage; and the surface may be a cylinder, a sphere, a torus, or something topologically more interesting. The cells might be connected to five neighbours, or more, or less, or form an irregular graph. Any number of dimensions can be used.

A continuous version might be a linear filter in which the speed of the interactive head depends on an output computed from the input read on the tape, and an internal state in the head.

Non-deterministic Turing machines are also possible, as are versions in which the head can cut and splice pieces of tape. However, once it is generalised too far, like any other model, it becomes something other than what was envisaged by the creator and is more of an outlook on computation than a particular device.

Exercise 3: Design a Turing Machine

You do not understand a virtual machine until you have written several programs for it. In this exercise, we try to understand Turing machines. One direct way to specify a Turing machine is a transition table. The columns are old-state, old-symbol, new-symbol, new-state, and movement, where L means go left, R means go right, and H means halt. The starting state is s, and the halting state is h.

For example, given a tape that contains only 0's except for a single run of 1's, we can make the run an even length by overwriting the first 0 on the right with a 1, if required. A Turing machine might do this by starting on the leftmost 1 and then moving to the right, switching between two states to keep track of whether the run of 1's is so far even or odd in length.

s	1	→	1	b	R
b	1	→	1	s	R
s	0	→	0	h	H
b	0	→	1	h	H

The machine switches between states s and b at each occurrence of the symbol 1 on the tape. Thus, s is the even parity state, and b is the odd parity state. When a 0 is found, the appropriate symbol is written and the machine halts. Even if the tape symbol is unchanged, we still fill in the entry in the table explicitly.

It is fairly easy to write a simple Turing machine simulator in a finite array, and it is recommended that you do so to help with these exercises.

The state of the Turing machine is easy to represent as a single finite string of tape symbols with an extra symbol, T, for the cpu; we assume that the final character on either end is repeated indefinitely. So, `00001T000` represents a tape full of 0's except for a single 1, and the Turing head is currently located on the cell containing the single 1.

The tally system of representing numbers just means to represent 1 as 1

and 2 as 11 and 3 as 111 and so on, with an arbitrary amount of padding with 0s on either side, so 0111000 also represents 3.

1. We write the addition 4+2 as `0001111011T000` in the tally system, where the T represents the Turing head initial position.

   Write a machine that moves only to the left, and changes this into a single equivalent tally base number.

2. In the binary system, we can increment a number by moving to the left, changing 1 into 0, until we find a 0. Write such a Turing machine.

3. Write a machine that converts tally to binary, by repeated use of an incrementing state on a binary number, and decrementing the tally number.

4. Write a machine that adds two numbers in binary. This time we include three symbols on the tape, so that the initial state might be `sss10010s101sss`, where s represents a space character. The desired output in this case is `sss10111sss`, where the original numbers have been erased.

5. Different Turing machines behave differently to the same input, such as an initially blank tape. Given a two-symbol tape, we can ask, how many 1's will Turing machine T write before halting (assuming it does halt). Since there are only a finite number of machines of a given number of states, there must be a maximum number of 1s that can be written. Such a maximal Turing machine is a *Busy Beaver*.

   Write a 1-, 2-, and 3-state Busy Beaver.

6. Determining that an n-state machine is a Busy Beaver, or just how many 1's an n-state Busy Beaver will write is very difficult, and the number rises rapidly, in the manner of Akerman's function.

   Can you work out any conclusions?

Notion 12: Non-Deterministic Machines

By an engineering definition, a *state* of a machine is some information which determines the machine's future behaviour. Strictly, it is tautologous that every state machine is deterministic. Any other machine does not have a state. It is unknown whether the universe as a whole has a state. But, even if a state exists, often only a part can be measured. The rest is hidden. Observing only the measurable part means that for each (observable) state there may be multiple possible futures.

For a discrete deterministic machine, each state leads to a unique next state. The next state is a function of the current state. Such a machine is characterised by a space of states equipped with a next-state function. It is natural to describe a non-deterministic discrete machine by a next-state *relation*.[5] Each state has a *collection* of next states.

Given a collection of (observable) states in which the machine might be, the set of states it might be in next is the union of the sets for each of the states in the collection. Thus, a non-deterministic machine is a special case of a deterministic machine on the power set of the states of the original machine. The final value returned by the non-deterministic machine is the value returned by the *first* deterministic machine to terminate. Of course, in general, this is a *collection* of values.

The power set of the states of a finite machine (see page 20) is also finite. So, a non-deterministic finite machine is still a finite machine, just on a larger state space. However, a finite machine viewed as a non-deterministic machine may be easier to design or modify than if viewed as deterministic. The non-deterministic machine has a special structure, admitting direct parallel implementation.

For Turing machines (see page 26), the machine state is the state and location of the cpu together with the state of the tape. This can be encoded as a finite integer. A Turing machine is a *countable state* machine. The set of subsets of a countable state space is uncountable, which takes us outside the scope of the modern desktop computer entirely. For this reason, the subsets will be restricted to be finite. The set of *finite* subsets of a countable state space is also countable.

[5] Sometimes referred to as a multi-valued function.

It is true, but not immediately clear, that a non-deterministic Turing machine can be emulated by a deterministic one. By using the Hanoi sequence (see note 22) we can splice a countable number of virtual tapes into one single tape. With one virtual tape used for scratch space, a deterministic Turing machine may simulate a non-deterministic Turing machine.

The true non-deterministic machine might compute more than, the same as, but never less than, the deterministic machine. Similarly, it is typically faster but never slower. However, this speed-up might be only in *time used*, since we really should count the steps of *every* one of the deterministic machines involved.

In practice, the overhead of emulating n machines on 1 is roughly proportional to n; the slowdown is about n as well. So it is a contradiction to have a greater than n times speed-up using a parallel machine. In practice, the speed-up may be much less. Logical dependencies between intermediate results computed during an algorithm can make it tricky at best to split it into parallel threads.

Many variations of this idea are used. In the Communicating Sequential Process [10] approach, the non-deterministic version acts in all ways that the hidden version can, but may also act in ways that the hidden version does not. A non-deterministic Turing machine might duplicate only the state of the cpu, becoming a shared memory device. A stochastic machine can be developed by introducing the probability that a machine will progress to a given state.

The full details of some of these models are complicated, and require great deals of theory to justify. However, the starting point of a set of states replacing a singular state is simple and foundational, finds many uses on its own, and has the merit of assuming very little about the nature of the non-determinism.

The question of whether there are problems for which a polynomial time algorithm exists on a non-deterministic Turing machine, but not on a deterministic Turing machine, is currently a (im)famously open problem in theoretical computer science (see page 146).

Exercise 4: Non-Deterministic Machines

A deterministic Turing machine can emulate a non-deterministic Turing machine (see page 30) and the principle is relatively straightforward. But the technical details require some work. Also, there is more than one way to complete the task. In practice, writing a Turing machine emulator would be a good idea before attempting this exercise.

Go through the details of making this work.

A couple of clues follow.

For the Hanoi sequence 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5, notice that every second entry is 1; and if you ignore 1's then every 2nd entry is 2; and if you ignore 1's and 2's, every 2nd entry is 3. In this way, we see that finding the elements of a specific tape is in principle straightforward. If you are on tape 1, skip 1 space, on tape 2 skip 3, and so on. But in order for this to work with a finite cpu, we need to store the info on tape. So we might put a tape number marker cell next to each virtual tape, but knowing which tape you are on cannot be stored in the cpu, or in a single cell (since there is an infinite number of virtual tapes). Of course, a Turing machine with an infinite number of tapes is different, like a register machine with an infinite number of registers. It could just use location 0 on each tape as an infinite random access memory.

A deterministic single-tape Turing machine simulating a non-deterministic Turing machine is ok, even for an infinite number of Turing machines, because we are simulating completely separate machines of which we have only a finite number at any one time, and each one has a finite description at all times. We could store the ones we are not using to the left, and fold the full tape over so that odd means negative and even means positive to get a full tape into one-half a tape to run the active Turing machine. However, this does require a lot of shifting machines around, and it does not enable us to determine which machine finishes first (once one machine finishes at n steps, how do we know that not one of the infinite number we have stops at, say, $n - 1$ steps).

Exercise 5: Quantum Computing

Ignoring the physics, a quantum computer inputs a complex vector representing qubit logic values which is multiplied by a hermitian matrix, which is then projected onto a subspace, and the magnitude gives the result. We will not duplicate this exactly. We take logic inputs as being just $(1, 0)$ to represent a logic 1 and $(0, 1)$ to represent a logic 0.

The following has the flavour but not the detail of designing quantum algorithms.

This is the identity logic function: $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$

This is negation: $\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$

This is exclusive-or:

$$\tfrac{1}{2} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$= \tfrac{1}{2} \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ -1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ -1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix}$$

The principle of operation is that the input logical data is presented in 01 or 10 form for 0 and 1, respectively. So one logic input is two dimensions, two logic inputs is 4, and so on. The output of the function is determined by taking the magnitude of the output vector, which must be either 0 or 1. The matrix must be symmetric.

Find as many of the 16 two-input logic functions as you can.

Find a system for determining matrices for logic functions.

Notion 13: Von Neumann Machine

The Von Neumann machine is traditionally described as being distinct from the Turing machine. It has a linear array of memory cells, each with a non-negative integer index, called its address. The memory array is read and modified by the central processing unit, or cpu. The cpu is an otherwise finite state-machine that can access several cells simultaneously. The location of the cells being accessed can be changed arbitrarily, a behaviour referred to as random access into memory.

But the memory array is simply half a Turing machine tape, and the cpu a Turing cpu. The multi-access aspect is identical to the multi-tape Turing machine in which the multiple tapes are just the same tape folded over and fed back into the cpu.

The mechanism for the random access is the pointer, which is a register that contains the address of a memory cell. However, unrestricted access to the pointer, as an indefinite precision integer, would completely change the character of the machine, and admit general-purpose programming within the cpu itself (see page 38).

In practice, the infinite memory array is addressed by a sequence of smaller moves, which could be simulated by having a stepping register in the Turing machine. The machine could then continue to step as long as the register was non-zero. Adding a finite capacity register to the Turing head does not change it from being a finite state-machine.

Thus, the Von Neumann machine is essentially a multi-tape Turing machine with a long-jump instruction to help speed things up. And, just as for the Turing machine, it is assumed that the Von Neumann machine memory begins with a finite amount of information, possibly in a finite number of non-zero cells.

The reasons for distinguishing the Von Neumann machine from the Turing machine are contingent, partly, on the politics of their invention. More significantly, the Von Neumann machine was designed to be implemented, and is the machine most implemented in desktop hardware, by a very wide margin. The Turing machine is simple and easy to implement, and while not intended to be used this way, would be perfectly

serviceable. But the limitation in movement to a single step is in practice very constraining, and easy to improve on in hardware.

A Turing machine storing its program in one part of its tape and the data in another has to move constantly between the two in order to determine what to do next, and then to do it. The ability of the Von Neumann machine to refer to two locations at once means that one can be the program and the other data, allowing the machine to execute program on data without the constant shuttling. However, this is almost identical to the 2-tape Turing machine, storing program on one tape and data on the other.

Further, the Turing machine has no problem with data overrun errors destroying the program. Memory protection mechanisms, both in hardware and software, are typically based around partitioning the memory into virtual distinct tapes in order to avoid memory clash.

The often-heard phrase *Von Neumann bottleneck* is a somewhat unfair reference to the fact that there are very few pointers into memory compared to the number of memory cells. In principle, if we could open this up so that a vast number of the cells could be processed at the same time, then we could speed up processing. But, in practice, (except in special cases often involving vector and matrix operations) there has been little prospect of an alternative. Further, if Von Neumann was not the most common hardware, we would no doubt hear about the register machine bottleneck, or the stack machine bottleneck, or something similar.

The fact is, the situation could be improved, but it is not easy to do so, and this is not specific to the Von Neumann machine. Ultimately the fact that the information needs to be sent across space would lead to the geometric bottleneck.

Technologically, the Von Neumann machine has been a very serviceable abstraction that has allowed the construction of practical, though conceptually "dirty", digital machines. The actual architectures are typically much more complicated to describe than is apparent in the above discussion, and involve many specific concepts, such as memory protection, indirection tables, memory mapped interaction, and so forth.

Notion 14: Stack Machine

Many abstract devices are expressed as a finite state machine cpu operating on a potentially infinite memory whose initial state contains only a finite amount of information. The volition is intrinsic to the cpu. The memory is used only as a scratch pad, remembering what the cpu cannot, analogously to a human computing with pencil and paper. But, curiously, it is in the memory that different devices differ, rather than in the cpu.

The symbolic stack machine memory is, naturally, a finite collection of stacks of symbols. Each individual stack is finite, but there is no finite upper bound to the possible stack size. The symbols are drawn from a constant finite alphabet. Each row in the table defining the cpu gives the current state, the stack to pop, the symbol popped, the stack to push, the symbol to push, and the new state. A halting state may be included.

A cpu augmented by a finite register storing a fixed-length integer is still a finite state-machine. The natural state space is the direct product of the states of the original cpu with the states of the register (see page 113). It is common to describe the nature of the cpu in the language of registers rather than directly in the primitive concepts of the state machine (see page 50).

To perform a binary operation the cpu can copy one symbol into an internal register first and then perform the correct unary operation on the other symbol. The register may be larger than the stack cells. Such a register can be pushed onto and popped from the stack, taking multiple cells and multiple steps. A common approach is to pop two arguments from a single stack, compute internally, and then push the result onto a (possibly different) stack.

A one-stack machine is limited by having to remember within its own state anything popped from the stack or lose the information forever. Such a machine cannot count the number of open brackets seen preceding a close bracket, and thus cannot compute whether brackets are matched. It is not a general-purpose computer. The simplest general-purpose case is the two-stack machine.

Stack machines have been implemented within compilers to convert infix expressions into postfix expressions, and within compiled code to evaluate the resulting postfix expression. Using an expression stack and a result stack, each expression symbol is popped in turn, and used to determine the action to take on the result stack. For example, a + symbol could cause the top two elements to be popped and their sum pushed. Data symbols in the expression are pushed directly onto the result stack. It is usually very easy to convert these actions into Von Neumann machine code.

Pure postfix numeric expressions do not need any brackets and are easy to evaluate. Repeatedly take the leftmost operator whose arguments are all known numbers and evaluate it. Reverse polish calculators are built on this principle. They typically have only one stack, the other stack is in the head of the person using the calculator.

The two-stack stack machine is almost identical to the one-tape Turing machine. The memory of the Turing machine is like two stacks of paper. The cpu views the top element of one of the stacks. To write is to pop and push a different symbol on the same stack, and to move is to pop and push the same symbol on different stacks.

In this analogy, the stack machine has the option of adding and deleting cells on the tape, which is one variant of the Turing machine. This approach has the philosophical advantage of needing only an Aristotelian *potential infinity* rather than the Cantorian *completed infinity*. More pragmatically, the entire Turing machine is at all times a finite structure. Related formal constructive definitions of Turing machines are neater than their infinite counterparts are.

Stack machines are a good basis to implement lambda calculus. Since the essential data structure in lambda calculus is a computed function, and the action is to call it, a stack machine can fairly naturally execute a lambda expression directly. This has been suggested as a practical way of building a computer. However, there are some issues with the building of large stacks, and the Von Neumann architecture is well established. There would have to be a major reason to change.

Notion 15: Register Machine

Instead of unlimited memory cells (see page 26) that store a limited range of integers, the register machine has a limited number of cells that store an unlimited range of integers. The register machine is analogous to the stack machine (see page 36); the stack is replaced by an integer. The standard constraint (starting with only finite information) is satisfied. The memory is a finite collection of finite integers.

The numeral `"123"` is not just a notation. It is an expression involving digits and an implicit operator. Let $a \circ b = 10a+b$. Then $(1 \circ 2) \circ 3$ is $(1 \times 10+2) \circ 3$, which is $(1 \times 10+2) \times 10+3$, which is $1 \times 100+2 \times 10+3 \times 1$. $123 = 1 \circ 2 \circ 3$ is an integer valued expression not a syntactic primitive.

The remainder from 123 after division by 10 is 3. That is, the digits of the integer do not disappear after the integer has been constructed, but can be determined arithmetically. An integer is a stack of base 10 digits. To read the top digit take the number modulo 10, to push a digit, multiply by 10 and add the digit.

More precisely, given b symbols to store, use

$$\begin{aligned} \text{push}(s, e) &= s \times b + e \\ \text{top}(s, e) &= s \bmod b \\ \text{pop}(s, e) &= s \operatorname{div} b \end{aligned}$$

An integer storing a collection of Boolean flags is the same situation.

From `step(n) = if n<0 then 0 else 1`,

form `m = step(n) * A + (1-step(n)) * B`

which sets m to two distinct values depending on the sign of n. This is one of a vast collection of useful arithmetic conditional constructions.[6] No Booleans need apply.

Two integer-registers equipped with arithmetic
allows general-purpose computing.

[6] Similar constructs can be useful C and Java.

Notion 16: Analogue Machine

Most of the abstract machines considered in this discussion are digital; they have discrete states that change discretely in time. However, in the earlier part of the 20th century, analogue machines were more common.

Analogue originally meant *by analogy*. Many physical systems can be modelled by differential equations. Two distinct systems having the same equations are *analogues* of each other. For certain differential equations, especially (but not only) linear ones, it is easy to set up a system of electronic components that behave as specified by the equations. A collection of such components constitutes an electronic analogue computer [13]. It could print out directly a waveform from within the system being modelled, something that could be very difficult to determine experimentally.

Digits are the symbols used in enumeration systems. Originally, computers that explicitly did arithmetic with digits were unable to compete with analogue computers for speed and adaptability. Such *digital* computers were promoted through their use in combinatorial problems such as breaking foreign encryption systems before and during the Second World War.

However, since the most obvious difference between digital and analogue computers is that the digital ones are discrete, and the analogue ones continuous, this has become the use of the word today.

An analogue computer is one that takes in a collection of continuous functions of time, and outputs likewise. The obvious application is in solving differential equations (but there are others). We feed in the forcing function for an ordinary differential equation, and the machine feeds out the solution to the equation. More generally, it might solve partial differential equations. There might also be auxiliary functions that are entered to assist the computation.

However, by interpreting ranges of real numbers as coding discrete symbols we can input and output discrete information. This is actually how digital computers are designed at a low level.

Notion 17: Cellular Automata

The input of a finite machine (see page 20) can be the output or state of another machine. Compounds built from a finite number of finite machines are still finite. However, the compound structure gives them an intuitive aspect that would not be apparent in a direct listing of the states.

The component machines of a *cellular automata* are typically arranged into a regular grid. Most often they are square 1-, 2-, or 3-dimensional (see page 42) but hexagonal grids are also common. At each clock-tick, every machine undergoes one transition, inputting the current state or symbol of the neighbouring machines. Less synchronous behaviour may also be considered.

Hexagonal grids have proved better than square grids for simulating physical systems such as two-dimensional fluid-flow. Artifacts of squareness often persist at a large scale, while the hexagonal grids discrete nature rapidly becomes unobservable.

Machines may also be built on other surfaces, such as spheres or toroids. However, regular grids on a sphere cannot be scaled. Irregular grids must be used. Common examples of simulated cellular automata are the numerical models for solution of partial differential equations. This merges seamlessly with finite element analysis.

By including states indicating *the cpu is here*, a one-dimensional cellular automata can easily emulate a Turing machine (see page 26). This leads naturally to a Turing Machine that wanders around on a surface. Often such a device is called an ant. One famous ant is Langton's ant — if on a white cell turn left, if on a black cell turn right, as you leave a cell, toggle the colour. It is famous for the complexity of behaviour it generates from such a simple rule, and in particular because it tends eventually to spend all its time building highways of regular pattern, no matter the (finite) pattern on which it was originally placed.

In biology, a developing embryo can be viewed as a solid cellular automaton able to change its underlying topology.

Notion 18: Unorthodox Models

This is a collection of special-purpose mechanical devices that appear to have an order of complexity advantage over the best digital techniques. They are selected partially for aesthetics, but also to emphasise the broad search space available for computational solutions. No discussion of the foundation of computation should ignore these options. Programming is about finding the desired computation within the given environment.

To sort a list of numbers, cut spaghetti into lengths proportional to the numbers (a linear time operation), thump them on a table (a constant time operation) and then pick the longest one, and the next longest, and so on (a linear operation). This is a linear time sorting routine.

To find the point whose weighted total distance to a set of other points is minimal, drill a hole through a board at the location of each points, put a rope through each hole with weighted proportionaly to the point, tie all the ropes together, and release.

To find out the shortest road network joining a collection of points, place a nail between two boards at the location of each point, dip into soap water. Shake it a bit to be more certain of a global rather than local minimum.

To find the convex hull of a set of points, drive nails into a board at required locations and stretch an elastic band around them. Alternatively, put a ruler against the nails (to find a single point on the hull) then wrap string, or just walk the ruler around. The ruler will make exactly one full rotation.

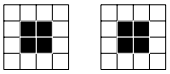
To find the shortest path in a network, make a string and beads model. The beads are nodes and the connecting strings have length proportional to the length of the corresponding edge. Grasp the points you want the shortest path between and try to pull the points apart; then the shortest path is the one that goes tight first. The longest path is found by constantly cutting the shortest path.

Notion 19: The Game of Life

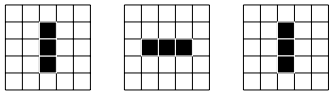
Although it is a specific machine, the modern desktop computer may be used to simulate a multitude of other architectures. Thus, an awareness of other virtual machines can be a boon to the practical programmer. Finite state-machines, for example, are often useful in the construction of protocols for security, cryptography, communications, or parsing of input. It has already been mentioned that a cpu machine has a compound of state machines as memory; however, the way in which these state machines receive their input and output means that they might as well be randomly scattered. An alternative is to stitch the finite state-machines together into a fabric called a cellular automaton. Cellular automata whose states are integers (or floating-point numbers) can be used for doing matrix multiplication in parallel (see page 232), in time proportional to the side, instead of the area, or the matrix. The numerical integration of partial differential vector equations is typically achieved using cellular automata.

A conceptually and historically significant exemplar is Conway's *game of life* (see note 8). Intuitively, Conway's life represents a population of stationary creatures analogous to coral or plants. In an infinite chessboard garden, each square either contains a plant, or it does not. At each clock-tick, every square is updated. If the square is occupied and has two or three occupied neighbours, then it survives into the next generation, otherwise it dies (of loneliness or overpopulation). If the square was vacant, but has three occupied neighbours, then a new plant is born into this square. The interest, however, is not with the individual cells, but the emergent phenomena.

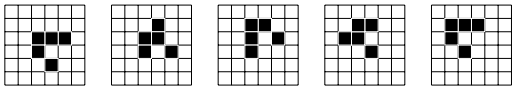
Some patterns are stable — they do not change under the transformation — for example, a 2×2 black square, the *block*. Any larger or smaller square is not stable.



Some patterns repeat themselves periodically — for example blinkers, which cycle between two states. (A row of three blinkers is called *traffic lights*.)



More interesting behaviour is shown by the glider. It moves. After four iterations it reappears in a different location looking the same.



Many moving patterns exist; gliders move diagonally, space ships move horizontally, some leave debris behind as they move. The speed of light is one cell per clock-tick.

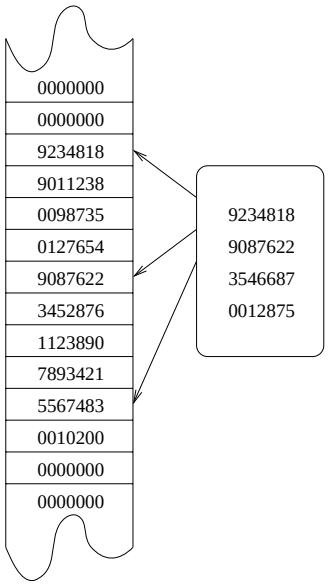
There are also *Garden of Eden* configurations, which have no precursor pattern, they must be explicitly entered into the grid.

The vast array of behaviours that this simple model produces is significant. In particular, there are very large structures with complex properties, such as reproduction and general-purpose computation. Some require millions of cells. Although the rules generalise in obvious ways, most variants tend to lead to uninteresting universes. However, two variants in particular, *high life* (survive — 2 or 3, birth — 3 or 6) and *night and day* (survive — 3 4 6 7 8, birth — 3 6 7 8) have been found to have interesting properties.

Can life evolve in Conway's life? The restriction to two dimensions is a severe restriction, sometimes thought to be completely prohibitive. However, some interesting work has been done on this (see note 9). Studies have also been made in higher-dimensional life (see note 11). Of central interest is the question of how to get the computer to recognise life forms, should they evolve. Rapid exact cycling is fairly straightforward to check. But recognising that something kept its general appearance is an open problem on which a lot of work has been done (in vision research).

Notion 20: The Modern Desktop Computer

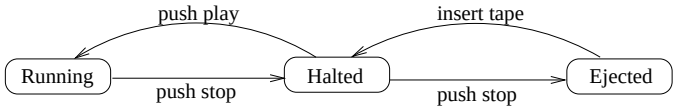
Almost all computers in use now, at the dawn of the 21st century, are cpu machines. A state machine (see page 20) called the cpu is augmented by a compound of state machines called memory cells that provide the input and record the output of the cpu. At each moment the cpu access to memory (see note 5) is limited in scope or in nature. But the resulting feedback expands the power above that of the cpu alone in a manner which may reasonably be emulated in our technology. Examples include the stack (see page 36) and register (see page 38) machines.



The standard modern desktop machine, regardless of brand, is inspired by the Turing (see page 26) or Von Neumann machines, in which the memory is a belt of cells, each of which contains a single symbol from a finite collection, and the cpu is a finite state-machine augmented by special pointers into memory. When the machine is started, the memory contains only a finite amount of information (see note 6). At each tick of the clock, the the memory is written, and the internal state of the cpu is updated. This simple cycle of computation is repeated indefinitely. Only a finite number of clock-ticks occur in a finite time. Only a finite amount of information is in the memory at any point in time. The machine, however, is potentially infinite. The memory and time available for the computation is indefinitely extensible. This subtle balance between finite and infinite has made this type of machine a very useful abstraction of computation.

Despite the utility of a potentially infinite model, any physical computer we can build has a finite number of states. Taking all the bits in the

memory (including disk storage) of a hundred million interconnected modern desktop computers, we still have less than `10^30` bits. So, the complete state of all these computers is a single finite integer in the range from `0` to `2^(10^30)-1` inclusive. If the machine does not suffer a hardware error, then, knowing the input (the key press or mouse click), the next state of the machine is determined uniquely by the current state. Thus, a finite list expresses the complete behaviour of the machine. Such an abstraction is called a finite state-machine.



A state machine may be in one of several conditions, called *states*. The state affects the way the machine responds. An example is an audio recorder with a red button that stops the tape if it is running but ejects the tape otherwise.

A desktop computer is a device that accepts a sequence of input events, acts in a manner dependent on its state, changes its state, and then awaits another input event. The number of states is astronomical, but finite.

The cpu of the desktop machine is a finite state-machine whose input is from, and output to, a potential infinity of information in memory. The design of the cpu-on-a-chip of the modern desktop computer is explicitly an exercise in the construction of a finite state-machine. The memory belt is implemented by the indefinite memory and disk resources. Many limitations proved for finite state-machines cannot be applied directly to the desktop machine because there is no hard limit to the resources (memory and time) available. It may be expanded indefinitely subject to vague economic and physical constraints.

Notion 21: Aspects of Virtual Machines

Digital machine: *a physical limit of discrete technology.* Although there are other things that may be said to compute, the basic idea of the modern digital computer can be nicely abstracted by saying that it is a state machine, whose states are countably many (may be numbered by the integers) and so are the transitions. Further, in any run of the machine, there are a countable number of clock-ticks.

Universal machine: *acts like every other machine.* It is possible to program some computers to simulate others. For example, you may be aware of PC simulators that were written for Macs, a number of years ago, in order to be able to run PC software on a Mac. I personally worked with a simulator that allowed Windows 3.1 to run under another operating system (RMX) as an application. In practice, any sufficiently complex computer can run a simulation of any other, with typically no more than a low-order polynomial slowdown.

General purpose machine: *to do anything that can be done.* It is a curious implication of the idea that any sufficiently complex machine can simulate another that the set of things that each can compute is the same. Indeed, there seems to be a natural boundary to digital computation. Problems that fall inside this boundary are called computable. All "reasonable" models of computation turn out to be able to compute exactly this set, the same set, no matter what type of digital computer we are talking about. There is a connection between this and constructive logic. There is a strong sense in which this limit is the limit of certain knowledge. It is the limit of those things that can be worked out, or known, in a finite amount of time, using discrete and definite steps.

Quantum computing: *A current research topic, but is it different?* Quantum computing is essentially analogue, a qubit encodes a continuous phase. The evolution of a quantum computer is related to multiplication of a vector by a matrix. This type of programming is difficult. The only good algorithms so far are of the Shor or Grover type. But it might be possible to implement these algorithms directly in nano-technology, and thus have a very fast solution to things that are currently outside the capacity of our computational technology. Whether a quantum computer can compute anything that is classically non-computable is

an open theoretical question. This is essentially a question about the relation between a continuous- and discrete-state machine. It might be possible, but it would require the precision of variable storage to be able to be increased without bound, making any solution physically meaningless. Any realisation to a bounded precision (and accuracy) would be equivalent to some digital machine. This is no more than the admission that a single real number contains an infinite amount of information, a fact that is of theoretical interest, but, due to finite signal-to-noise ratios, of no practical consequence.

Implementation: *Can we build it?* The idea of a finite state-machine is powerful and forms the heart of all our computational machinery. The memory structure used in each of the different digital computers has a potential infinity of information. This allows us to implement limited extensible versions of many types of machine, with the idea that it will run all those programs that do not require more than a certain amount of information storage. All the various discrete-state machines mentioned in this book have been implemented in this sense.

The register machine is essentially the same as the stack machine in realisation, and the Von Neumann machine has the advantage of speed and cost in most implementations. This would be reduced by the economies of scale, or the use of different compilation technology. Along these lines, multi-stack machines have been suggested and built which use lambda calculus as machine code.

Analogue computing was more common in the first half of the 20th century. But analogue signals are inherently hard to correct, while digital signals can be checked and corrected. In addition, digital computers can be easily adapted to other tasks, such as symbolic processing. These two factors lead to the demise of the analogue computer.

Quantum computing is in many ways a return to analogue computing, based on the behaviour of certain nano-technology. But, researchers are having a very hard time implementing anything non-trivial in quantum technology.

Notion 22: Aspects of Programming

Programming: Many abstract machines (see page 18) have a state which can be divided into a product of a processor state and a memory state. The pure use of this machine is to encode the input value into the memory and start the processor in its distinguished starting state. Later the output value is retrieved from memory. This encodes a function $F(problem) = solution$. But a new machine is required for each new class of problem. An alternative is to allow another part of memory to be used to record information that tells the computer how to solve the class of problem it has been given. In digital computers, this extra information is a sequence of symbols. In an analogue computer, it might be the interconnections between the analogue elements. This extra information is called a program. Technically this means that the problem is specified as the union of the two sections of memory, but the outlook is very different. $F(program, problem) = solution$. Given the machine, the task is to find the program that will solve the problem. We view each program as producing a new virtual machine.

Real programming: Programming is not menu-lookup. Certain program code can be written by checking which library call performs the required action and providing the correct arguments. This is not the type of programming discussed in the book, and in fact does not satisfy the intention of the word as used, except in a trivial sense. Real programming means to increase the computational capacity, to begin with a set of operations, and develop them into new operations that were not obviously implicit in the original set. To borrow an economics term, real programming is computational value-adding, i.e., providing new capacity in a non-trivial manner.

Algorithmic computation: In the strict sense, an algorithm for determining a result is a process that has discrete, exactly defined steps and is sure to terminate after a finite number of steps, producing as a result the required result. Precisely when a step is exactly defined is actually a philosophical matter, but in practice it appears that exactly defined means expressible by formal, unification production (see page 64), axioms. For an ongoing process that has no natural termination, the requirement is that the steps are exact and correctness is certain.

Heuristic computation: When no algorithm is available, or those that are available are too slow, a heuristic might suffice. A heuristic generally has precise, uniquely determined discrete steps, but it is not certain to produce the right result. It is often considered acceptable for a computer to be right only a certain fraction of the time if this results in a lower use of resources.

Stochastic computation: An alternative approach related to heuristics is a computation in which random choices are made at certain points. The curious character of these computations is that they can be developed so that there is a tweaking parameter (such as how long a loop runs) that can push the probability of a correct answer toward certainty. Once a process produces the right answer with a chance of 1,000,000,000:1, we would start to worry about hardware failure before worrying about the process going wrong. Many computations in theorem proving, group theory, prime factorisation, and cryptological work are stochastic these days.

Another curious character of stochastic computation, in practice, is that the asymptotic correctness is established on the assumption of the use of a true random number. But, in a typical computer, this is implemented as a pseudo-random number, generated by a computation with extremely erratic, but deterministic, behaviour. The resulting patterns can break the computation. In particular, some attacks on certain encryption systems involve locking onto the pseudo-random computation. This would not be possible if the encryption was based on a truly random number.

This is a pragmatic computational problem. But it leads us immediately into the deeply philosophical problems of what a random number is, and whether there really is such a thing. One suggestion for getting around this is to use physical noise sources, especially ones based on quantum mechanical principles. However, whether that will remove the correlation with other factors remains to be seen. A device sitting inside a computer is subject to many influences and might not be random even if the truly isolated device would be. Perhaps randomness in isolated quantum systems actually comes from that isolation, leading to true lack of information.

Notion 23: Register Indirection

A typical contemporary desktop computer is built on a collection of low-level storage cells. Each cell contains at each instant a single symbol from a finite alphabet (see page 44). Each cell is referred to via an address which is unique to that cell. Each of these identical cells is called a *register* (see note 15). We assume that the address space and the alphabet are the same (see note 14) so, any register may be interpreted to contain the address of another register.

It is common to use numerals from 1 to m as the symbols. Parentheses refer to content. So (123) means the symbol contained in the register with address 123, or more briefly (123), is the contents of 123. If x and y are symbols, then $x \leftarrow y$ is the command to place the symbol y in the register at address x. This is similar to assignment in many common languages, but there is no implicit indirection. What you see is what you get. This is best understood by interpreting all symbols in register language to be *constant* symbols. That is, $x \leftarrow y$ does not affect x any more than `M[1]=5` affects the number 1. There are no variable symbols in register language. Everything dynamic is stored in the registers.

In C, a variable symbol is a constant pointer, an address. The assignment `x=y` is implemented as $x \leftarrow (y)$. Implicitly the symbol y is treated as an indirect reference to its contents. So if x has (or is) the address 123 and y is the address 64, then `x=y` in C causes the contents of location 64 to be placed into location 123. In register indirection, this is $123 \leftarrow (64)$. The command $x = y$ has one level of indirection on the right, and none on the left. The automatic de-reference can be prevented in C by using the form `x=&y`, which means $x \leftarrow y$. The `&` quotes the variable name. In C, `*x` is largely the same as (x) in register indirection. In C, $*x = y$ is $(x) \leftarrow (y)$.

Any occurrence of the symbol x in C means (x). C expects an l-value[7] to have parentheses, and removes them. Sometimes an error occurs, *l-value expected*, exposing this behaviour. In register indirection, $123 \leftarrow 5$ is valid, but in C, $123 = 5$ generates an error because the compiler tried to remove one level of indirection. This is the meaning of `&`. The term $*x$ means (x) but `&`$*x$ means x. Thus, `&&`x is often an error. A

[7] Left hand side of an assignment.

constant value has no address. But $**\&\&x$ has the intuitive meaning of x, even though by strict application the second `&` was an error. Some C compilers allow compound expressions such as $**\&\&x$ with any number of $*$ and `&`. Occurrences of `&` and $*$ cancel. This is useful in association with `#define` directives.

Each machine code instruction has an address. The program counter pc is a register that stores the address of the current instruction. Thus, $pc \leftarrow 123$ is a jump. Also if sp is the location of a stack in memory, then $(sp) \leftarrow (x); sp \leftarrow (sp) + 1$ pushes the value of variable x onto the stack, while $sp \leftarrow (sp) - 1; x \leftarrow (sp)$ pops the value from the stack. This introduces $(sp)-$, which is a post-decrement operation. Obtain the value and then decrement the register. So the pop is $x \leftarrow (sp)-$ and the push is $+(sp) \leftarrow (x)$.

Something very similar was used on the PDP series of machines, and it is likely that this is where C obtained its `++` and `--` operators.

Register indirection is supplemented by at least one conditional form, such as "f x then $x \leftarrow 23$". With only these elements in the language it is typically possible to give a simple formal specification of each assembler instruction in a machine. The C language can also largely be defined in these terms.

Register indirection is a good concrete model for an understanding of the operation of pointers. All abstract pointer algorithms can be cleanly expressed in this manner. It is a high-level generic method for understanding many specific Von Neumann machines, and machine codes.

Caveat: it is common to find register language in which $(x) \leftarrow y$ means *the contents of x becomes y*. That is, what we are notating as $x \leftarrow (y)$. This leaves the requirement for getting at the actual value of y, which is done by an *immediate* reference, denoted typically $\$y$ or $\#y$.

Notion 24: Pure Expression Substitution

Given that $x = 6$, we deduce that $x + y = 6 + y$ by replacing the symbol "x" by the symbol "6" in the expression "$x + y$". This replacement of equals by equals is deeply fundamental to all formal reasoning and may be performed in an entirely mechanical manner.

The environment of an expression such as "x+y" is the set of values of the symbols it contains. Not all symbols need to be given a value. A symbol is *bound* if it has a value, and *free* if it does not. An environment is a partial function whose domain is the set of symbols. An environment may be naturally represented as a set of ordered pairs. Thus, $\{(x, 5), (y, 6)\}$ acting on "x+y" produces "5+6". The value might also be an expression, in the environment $\{(x, (a+b)), (y, 23)\}$, the expression "x+y" becomes "(a+b)+23".

An environment $E = \{(x_i, y_i) : i \in I\}$ is a function taking a primitive expression, and returning an expression. The action A of E is a natural extension of E to compound expressions.

1. $\forall i \in I : A(x_i) = y_i$
2. $\forall x \notin \{x_i : i \in I\} : A(x) = x$
3. $A(a_1, .., a_n) = (A(a_1), .., A(a_n))$

Although it is non-trivial to prove, A is a function. As a set of ordered pairs A includes composite expressions on the left, if we allow composite expressions in E, then the action may become a more general relation than a function. For example, starting with $E = \{(xy, w), (yz, w)\}$ we get $A(xyz) = wz$ or xw.

A substitution system $S = \{(x_i, y_i) : i \in I\}$ is a set of ordered pairs of expressions. A substitution is a relation. The action of S is the closure of S under composition of its elements.

1. $\forall i \in I : (x_i, y_i) \in A$
2. $\forall x \notin \{x_i : i \in I\} : (x, x) \in A$
3. $\forall (a_1, b_1)..(a_n, b_n) \in A : ((a_1..a_n), (b_1..b_n)) \in A$

Generically, an inductive definition of a set will include a number of clauses of the form *if x,y,z are elements then C(x,y,z) is also.* From this

we derive the induction if A includes $(x_1, v_1), .., (x_n, v_n)$, then A also includes $(C(x_1, .., x_n), C(v_1, .., v_n))$.

The action of an environment is a special case of a substitution system, a simple substitution system. If $S_1 = \{x_i \rightarrow y_i : i \in I\}$ is a simple substitution and S_2 is any substitution system, then relation composition $S_1(S_2(E)) = \{(x_i, (S_1(y_i)) : i \in I\}$. This is a fairly natural result but is not generically true. If compound expressions exist on the left of S_1, then it is possible for a rule in S_1 applied to $S_2(xy) = S_2(x)S_2(y)$ to match on a subexpression that overlaps the two pieces. But this does not occur in the case of a simple substitution system, derived from using only primitive symbols on the left.

The composition equation was determined using the assumption that the left-hand sides were primitive symbols. The rule does not cover all cases with a composite left-hand side. But if there is no instance of an entities on the left overlapping, then it does still work. Intuitively we could replace each instance of the composite symbol by a single primitive one.

Each pair (x, y) stands for a logical equality. One substitution is *more general than* another if, treated as a collection of equalities, the latter implies the former. A term is *primitive* with respect to a substitution if the substitution leaves it unchanged.

By allowing a substitution to be applied in the reverse as well as forward sense, we can use it to generate a concept of equality very similar to a Prolog program. If, however, we allow substitutions to be applied only in the forward direction, then we have a model of computation very similar to a Haskell program.

One aspect not covered here, but of significance in lambda calculus, is the concept of locally bound variables. If our expression is $(sum(n = 1to10)n) * n$, then substituting $n = 2$ would produce $(sum(2 = 1to10)2) * 2$, which is not quite what we intended. The n used as an index for the summation is distinct in intention from the free occurrence of n. The idea of substitution can be modified by allowing local bindings to be recognised, and to avoid substituting in this case.

Notion 25: Lists Pure and Linked

A singly linked list allows us to put something on the head of the list and find it there again later. Abstractly it is identical to a stack. To look down the list we have to move past all the earlier elements. An abstract list datatype is a tuple (List,Item,head,tail,cons,empty), such that —

empty is a List
for each Item a, and List b : head(cons(a,b)) = a
for each Item a, and List b : tail(cons(a,b)) = b

This can be constructed by the following substitutions:

empty = ()	By inspection ...
head((a,b)) = a	head(cons(a,b)) = head((a,b)) = a
tail((a,b)) = b	tail(cons(a,b)) = tail((a,b)) = b
cons(a,b) = (a,b)	So, the axioms are satisfied.

The (Haskell/Prolog style) list [1,2,3] is (1,(2,(3,(())))).

It is good, if possible, to implement an abstract type so that there is a natural correspondence between the code and the axioms. This helps to prove, technically and intuitively, that the code conforms to the axioms.

Firstly, the type definitions:

```
struct pair {item x; list *next;};
typedef pair *list;
pair newPair(item x, list l){pair p={x,l}; return p;}
```

Now the definitive code:

```
item empty = 0;
item head(list l){return l->x;}
item tail(list l){return l->next;}
list cons(item x, list l){return newPair(x,l);}
```

This code is so close to the substitutions that it is almost just a syntax change. Thus, we have a rigorous implementation.

Pragmatically, these axioms are incomplete.

Firstly, there is no way to compute head(empty) and tail(empty). In principle, the attempt to do so is an error, and we could insist that any system that satisfies the axioms must assure that these terms are never evaluated. Technically, this is a valid solution, with rigorous logical justification. But in practice, it puts too heavy a load on the user of the abstract type. What we need is error handling. A simple option is ... head(empty) = tail(empty) = error = head(error) = tail(error). The full study of pure error handling is non-trivial.

Secondly, they are satisfied by this trivial construction:

head(empty) = empty
tail(empty) = empty
cons(empty,empty) = empty

The generic way out to state that the system must not satisfy any equality involving head, tail and cons, other than those implied by the original axioms. This amounts to such things as cons(a,b) $\neq$ empty. But negative axioms are more difficult to deal with.

Pure expressions are naturally interpreted as trees. The nodes are weighted by operators, and the subtrees are arguments. Recursion scans down the tree. But the only way back is to return from the recursion. Generic networks have no natural counterpart in pure expressions. A network can be a list of pairs of nodes, but a constant time search for neighbours does not exist. Contrast this with back pointers in doubly linked lists. Or, perhaps, it does not exist for pointers either; rather, pointers are fast hardware assistance for random lookup limited to the machine memory. Severe slow-down occurs once disk or network storage is required. Pointers are a fast hardware implementation of a network. With an in-built fast network type included, pure expressions can match the impure performance. Haskell, which is very pure, has array and network types available if required.

However, reorganisation of the details of the software can often achieve the same aim. The programming required in the pure case is for sorting, for example, is different, rather than less efficient.

Notion 26: Pure String Substitution

Pure substitution into expressions (see page 52) respects the bracketing. Acting on $x \times y$ with $\{(x, a + b), (y, 23)\}$ produces $(a + b) \times 23$, not $a + b \times 23$. The latter, using implicit bracketing, would standardly be interpreted as $a + (b \times 23)$. Expression substitution software based on trees would not typically make this sort of mistake. But it is a very common mistake among teenagers learning algebra in secondary school. Partly, this is because the student is dealing with the expression as a pure string, an otherwise unstructured sequence of characters on the page. If we literally replace "x" by "$a + b$" in the string "$x \times y$", then we get "$a + b \times y$", resulting in a change in semantics.

With this rather dubious beginning,[8] we enter the world of pure string substitution. We can act on a string s using a pair of strings (s_1, s_2). Find a decomposition (by catenation) of $s = as_1b$, and construct the string $s' = as_2b$. We have replaced an instance, in s, of the substring s_1 by an instance of s_2. The decomposition might not be unique, so there are multiple possible resulting strings. String substitution does not inherently respect the structure of the expression that the string might happen to represent.

String substitution is a special case of replacing one directed subgraph by another (see page 117). In the simplest case, replace a subgraph by another with the same number of terminals. In the string case, we replace a compound directed link by another in the same direction.

A string is *primitive* with respect to a substitution if the action of the substitution is sure to leave it unchanged. For example, $zxzx$ is primitive with respect to $xx \to y$. A string reduction machine repeats the action of a substitution on a state string until a primitive string is produced. The string machine includes in a natural way parallelism and non-determinism. Two substitutions working on non-overlapping pieces of the string may act at the same time, when there are multiple options at one location a non-deterministic choice is made.

[8] I have found that often advanced study in mathematics or computation is a casebook of things you were told not to do in secondary school.

A string reduction machine is a general-purpose computer. The substitutions on the right define an operator I that increments a binary number enclosed in brackets. The use of c forces the computation to be local.

```
0c → 1
1c → c0
[c → [1
]I → c]
```

An example shows its action. Although the brackets do actually delimit the binary number, this is an emergent property; in detail they are treated as any other character. DNA computing works this way. The string of symbols is the sequence of bases, and the substitutions are performed by the various enzymes.

```
  [100111]I
→ [100111c]
→ [10011c0]
→ [1001c00]
→ [100c000]
→ [1010000]
```

The basic notion above, of a symbol filter, is broadly applicable and corresponds to the mechanism of DNA computation. Enzymes move along the DNA performing operations, such as hydration. As the desired computation becomes more complex multiple symbol filters are required, leaving the possibility of one filter operating on another to change its function.

To show string reduction is general-purpose, construct a Turing machine. We could use a symbol for each state of the Turing head (see page 40), but the state is naturally indicated by `[xxxx]`, where the `xxxx` is a binary sequence, the state number. We take the current tape symbol to the be one to the left of the state, and have substitutions of the form `0[001]`→`[010]1`, to indicate a choice of move to the left and `0[001]0`→`01[010]` for a move to the right.

The lack of respect for bracketing and variable symbols can cause trouble. How to deal with parameterised computation and function calls? How would we design a lambda reduction engine on a string reduction machine? It can be done. The problems can be solved. To solve it once is an excellent programming exercise for honing general programming skills.

But, commonly, a non-trivial level of respect for brackets and variables is assumed in working with reduction machines, largely because it is tedious and otherwise unilluminating to achieve from first principles each time we want to program with strings.

Notion 27: The Face Value of Numerals

A distinction is sometimes stressed by expositors between the abstract concept of *number* and its written form *numeral*. This emphasises that the numeral has an existence in its own right, but obscures the fact that number and numeral can validly be equated. The identification of written form with semantics is natural in computation. If we define the non-negative integers to be *literally* the strings `0`, `1`, `10`, and so on, which are normally taken as a *representation* in binary notation, then we may define increment as an operation on strings by the following axioms:

$inc(0) = 1,$
$inc(1) = 10,$
$inc(X0) = X1,$
$inc(X1) = inc(X)0,$

The symbol X (and any other capital letter) stands for any nonempty string, and string concatenation is represented by juxtaposition of symbols.

$$inc(111) = inc(11)0 = inc(1)00 = 1000$$

Technically we may then say that the abstract non-negative integers are the set of all such systems isomorphic to this. But this leads to logical problems with the term *all*. The approach used here avoids that, and is also similar in spirit to the Von Neumann style encoding into set theory that was used to give pure set theoretical structures for various abstract structure in mathematics.

Extending the identification of syntax and semantics, we interpret *inc*, not as a function, but as a literal string. The entire computation is an expression being reduced by the unification reductions.

```
   inc(X0) → X1
   inc(X1) → inc(X)0
add(X0,Y0) → add(X,Y)0
add(X0,Y1) → add(X,Y)1
add(X1,Y0) → add(X,Y)1
add(X1,Y1) → add(X,inc(Y))0
 mul(X,Y0) → mul(X,Y)0
 mul(X,Y1) → add(X,mul(X,Y0))
```

Arithmetic is expressed completely by the reductions. The inclusion of the terminal (not involving a variable) cases is left as an exercise.

Given a set of axioms, an irreducible element is one which does not match the left-hand side of any of the axioms. The meaning of a string can be taken as the irreducible element to which it reduces. Of course, for arbitrary axioms there is no certainty that the irreducable element is unique, or even that it exists. However, certain systems (such as lambda calculus) have been shown to have this property, known as confluence. A unique irreducible element is sometimes referred to as a normal form.

If every sequence of reductions ends finitely at an irreducible element, the set of axioms are said to produce a Noetheren reduction system. The property that one string can be modified into another by a sequence of substitutions is sometimes used as a definition of equality within sets described by a language. The pragmatic problem is that determining this equality is not computable in general. One significance of a normal form in a Noetheren reduction system is that its universal existence means that equality is a computable property with a simple algorithm. Just reduce each string to its normal form, and see if they are the same.

An example of this is the expansion of algebraic expressions using only addition, subtraction, and multiplication. By multiplying out all the terms, we obtain pure powers of the variables, which can be listed in a systematic order to determine the equality of two such expressions.

It is significant for our interpretation of strings that we recognise brackets (see page 52). If not, then when we try to compound the expressions, as in $add(10, mul(110, 1010))$, the rule might misfire to set $Y =$"1010)", leading to a syntactically incorrect irreducible. The problem is partially relieved but not solved by insisting that only the whole expression can match. However, if we insist that each variable can only match syntactically correct substrings, then the problem is solved.

But if this approach is too high-level for the reader, rest assured that it is possible, by selecting the reduction rules appropriately, to produce systems that operate correctly, without *any* special interpretation of the syntax. With this in mind a simpler set can be used, $inc(X0) \to X1$, $inc() \to 1$, and $inc(X1) \to inc(X)0$, involving the form $inc()$, which can be thought of as standing in for the increment of an empty string, or just a working string as part of the computation.

Exercise 6: The Face Value of Numerals

This is a non-trivial, but very instructive, exercise in digital algorithmic thinking. It is something (like checking the Jacobi identity for the vector cross product in mathematics is for mathematicians) that every computer scientist should do once in their lives. Thinking of a binary integer is a string of 0's and 1's, develop some arithmetic. Encoding numbers as pairs (see page 215) is useful here.

1. Develop syntactic reductions for equality, increment, addition, integer division, multiplication, modulus, order, and greatest common divisor for unsigned integers.
2. Develop syntactic reductions for equality, increment, addition, subtraction, modulus and order of signed integers.
3. Develop syntactic reductions for division in the rationals
4. Develop complex rational arithmetic.
5. Some people say quantum computers can avoid the limitations (see note 23) of reduction.

   What do you think?

It is recommended that a simple program to reduce expressions given a set of reductions be written in conjunction with this exercise. Brute force search for the substring and copying the whole string during substitution (for example using `sprintf` in C) is probably sufficient, but there are many more-sophisticated algorithms that can be used.

The basic concept of substituting one part of a structure for another is not a side issue in digital computing, but really the central theme.

Macrolanguages (short for *macroscopic substitution*) use essentially the type of approach as above.

Exercise 7: Pure Substitution Computation

Implement binary arithmetic with pure string substitution. The solution is a set of substitutions (see page 56) that inevitably reduce the valid input string to the required output string. It is highly recomended that you write a string reduction engine (in your favorite language) prior to this, so the solutions can be tested and debugged.

Each part introduces a filter symbol, intuitively similar to a Turing cpu, which overwrites the tape as it moves (but can also splice in new cells). The whole computation is a non-deterministic Turing machine. A stationary symbol might generate several filters that move away from it to perform one each of several stages. Binary numbers are represented as [0010110], or [bbabaab] to keep two numbers distinct.

1. Create `A`, which changes 0s to *a*s and 1s to *b*s,
   For example, `A[01001]` becomes `[abaab]`.
2. Create `B`, which interlaces a 0-1 number with an a-b number
   For example, `B[10011aabab]` becomes `[1a0a0b1a1b]`.
3. Create `S`, which adds interlaced 0-1 and a-b numbers
   For example, `[1a0a0b1a1b]S` becomes `[11000]`.
4. Combine 1,2 and 3 into `+`, which adds two numbers
   For example, `[10011]+[00101]` becomes `[11000]`.
5. Create `-`, which substracts two numbers
   For example, `[101]-[10]` becomes `[11]` .
6. Write a multiplication operator $*$
   For example `[1010]*[10]` becomes `[10100]`.
7. Write `P`, which reduces prefix arithmetic expressions with addition, subtraction and multiplication to a single binary number. (Protect each operator from the action of the others).
8. Write `B`, so that is is similar to part 7, but uses infix arithmetic and recognises parentheses `"("` and `")"` correctly.

Notion 28: Solving Equations

Technical programming from a formal specification means finding code that is the solution to an equation in unknown code symbols. The specification is the equation the code must satisfy. The concept of solving an equation applies in a very broad context. Computation itself is repeated solution of formal equations. The constraint on the code is an expression in constant and variable symbols.

To solve is to find values that when substituted for the variables produce a true equality. Typically, in $a + b = 12$ the variables are $\{a, b\}$, and constants are $+$ (a known binary function) and 12. The $=$ symbol is metalogical.[9,10,11] To solve $a + b = 12$ is to find values for a and b. The substitution $\{a \to 4, b \to 8\}$ acting on $a + b = 12$ yields $4 + 8 = 12$ which is true. So this is a (*not the*) solution to the original equation.

This is the same for an unknown function.

We solve $f(x + y) = f(x)f(y)$ for f as follows:

$f(x) = f(x + 0) = f(x)f(0)$ so either $f(x) = 0$ or $f(0) = 1$.

$f(nx) = f((n - 1)x)f(x) = f((n - 2)x)f(x)f(x) = \cdots = f(x)^n$

$f(m\frac{x}{m}) = f^m(\frac{x}{m})$ so $f(\frac{x}{m}) = f(x)^{\frac{1}{m}}$

That is, $f(ax) = f(x)^a$, and so $f(a) = f(1)^a$ for all rational a. (This extends to complex a if we assume continuity). The formal substitution $\{f(x) \to b^x\}$ acting on $f(x + y) = f(x)f(y)$ produces $b^{x+y} = b^x b^y$, which is true. The solution is generic. (This equation is definitive of exponentiation).

A *general* solution is a description of all the solutions, often couched in terms of expressions over some free-ranging variables. In the above case, this could be $\{a \to t, b \to 12 - t\}$, in which t is a variable taking values from a numerical set, which might be whole, real, or complex numbers.

[9] The distinction between proof and truth.
[10] It means, *If you don't already know what I mean then, I cannot explain it.*
[11] It's a Zen thing.

Solving multiple equations, such as $\{a+b=12, a-b+c=10\}$ is a similar process, we look for values for all the variables involved that when substituted yield a collection of true equations. In this case, a specific solution is $\{a \to 3, b \to 9, c \to 16\}$, and a general solution is $\{a \to t, b \to 12-t, c \to 22-2t\}$, where t is a free parameter. More generally there might be more than one parameter.

Collecting terms on one side produces $\{a+b-12=0, a-b+c-10=0\}$. The solution is equivalent to finding values of a and b such that *all* the expressions in $\{a+b-12, a-b+c-10, 0\}$ become equal. The substitution is said to *unify* the expressions. Even without arithmetic, any set of equalities can be expressed as unification by forming tuples. For example unify the set, $\{(a+b, a-b+c), (12, 10)\}$ to solve the original equations.

In C, we define a function `int f(int x, int y){...}`. When calling f, in `a=f(23,45)`, the set $\{f(23,45), f(x,y)\}$ is unified to determine the values x and y take as local variables in the body of f. Commonly the syntactic solution $\{x \to 23, y \to 45\}$ is used. But if we know that $f(x,y) = f(y,x)$, a second solution, $\{x=45, y=23\}$, is obtained. This is unification with axioms. Unification with axioms is the generic concept of solution of equations.

For commutative f, an optimising compiler might use the second solution for reasons of size or speed. The syntactic (or empty axiom set) solution has the significant property that it works regardless of the nature of f, while other solutions require more information.

Computation is a process of solution of equations. Syntactic solutions, found by unification, exist independent of meaning of the symbols. We determine values for the variables which make the expressions equal. A general solution may exist with free parameters whose value is arbitrary (within the context). The complete solution set depends on the nature of the constants. But, a solution with less axioms is also a solution for the original. Solutions that depend on no axioms work by making the expressions syntactically identical. This is known as a formal solution.

Notion 29: Pure Unification

Definitively, a formal equation is solved without any reference to any semantic domain for the symbols. One solution, for x, y, a, and b, to the formal equation $m(x, y) = m(a, b)$ is the substitution (see page 52) $S = \{x \to t, y \to t, a \to t, b \to t\}$. Applying S here produces $m(t, t) = m(t, t)$, true regardless[12] of the semantics of m. The substitution unifies (see page 62) the set $\{m(x, y), m(a, b)\}$, making all the expressions identical. More formally, a substitution, S, unifies a set, E, of expressions exactly when $\forall e_1, e_2 \in E : S(e_1) = S(e_2)$.

The input (E, V, C) to unification is a set E of expressions over a set $S = V \cup C$ of symbols, paritioned ($V \cap U = \{\}$) into variables V and constants C. A substitution over V is called an environment. The task is to find an environment that maps E to one single expression. There is typically more than one solution.

An environment S_1 is at least as general as S_2 if there exists S_3 such that $S_2 = S_3 \circ S_1$. Generality is a measure of how little the variables are constrained. The substitution $\{x \to a, y \to b, z \to c\}$ is less constraining than $\{x \to a, y \to b, z \to b\}$, which is less so than $\{x \to a, y \to a, z \to a\}$. A unifier S_1 of E is a *most general unifier* if for any other unifier S_2, S_1 is at least as general as S_2. All the most general unifiers are essentially the same, differing only in the choice of free variable names.

For the example above, $S_1 = \{x \to p, y \to q, a \to p, b \to q\}$ is a most general unifier, and so is $S_2 = \{x \to w, y \to z, a \to w, b \to z\}$. The original solution $S = S_3 \circ S_1$, where $S_3 = \{p \to t, q \to t\}$.

To unify $M(x, (a, b))$, $M(F(a), (a, z))$ and $M(F(F(b)), (y, b))$, over $\{a, b, x, y, z\}$ we *need* $x = F(a)$, $a = F(b)$, $y = a$ and $z = b$. Lining up the expressions makes this clear.

```
M(x         ,(a,b))
M(F(a    ) ,(a,z))
M(F(F(b)) ,(y,b))
```

By using $a = F(b)$ in other equalities involving a, some reflection should convince that $\{x \to F(F(b)),\ a \to F(b),\ y \to F(b), z \to b\}$ is a most general unifier. But if we included $M(b, (a, b))$, then we would have needed $b = F(F(b))$, for which there is no finite formal solution.

[12] Well, not quite (see note 10), but the conclusion is very broadly sound.

Functions might not be constants, so we use s-expressions and denote $f(x)$ by (f, x). To unify $\{(f_1, x_1), (f_2, x_2)\}$ is to simultaneously unify $\{f_1, f_2\}$ and $\{x_1, x_2\}$. The natural problem is to find an environment that simultaneously unifies each set in a collection of sets of expressions, such as $\{\{f_1, f_2\}, \{x_1, x_2\}\}$. One level of computation on each set of expressions produces a collection of sets of expressions. The union of these collections gives an equivalent collection to unify. When a variable symbol occurs in a set of expressions, as in $\{x, f(a), f(f(b))\}$, we may set aside a pair, such as $(x, f(a))$, as giving the definitive restriction on the variable symbol.

Some expression must be of maximal depth, and at each step we break up compound expressions, removing one level. So, eventually all sets contain a primitive symbol. We remove any instance of $x \to x$ and if x has multiple values, we select one, and unify the rest. We eventually have only symbol–value pairs. A dependency exists when x_1 occurs in the value of x_2. If x_1 depends on x_1, or on x_2 that depends on x_1, or there is any longer chain arriving back at x_1, then a loop exists, such as $x \to f(x)$. If there is a loop or a constant is given a compound value, then no finite formal unification is possible. Otherwise, this is an environment that unifies the original expressions. Treating the data as a collection of equalities, the information content is unchanged. The resulting environment is a most general unifier.

We don't have to wait for the complete expansion to detect problems. In actual implementation, the *not-unifiable* result can be returned as soon as a problem is detected.

However, loops are not illogical, and allowing them then the above process does obtain the minimal extra conditions for unification. If we accept these extra conditions we obtain unification. Unification with axioms starts with similar extra equalities. The above algorithm finds the minimal set of axioms required. Unification with axioms does not always admit a most general unifier.

An axiom is a triple (E_1, E_2, V) of two expressions and a set of variables. I stands for all the subexpression reductions $S(E_1) \to S(E_2)$, where S is an environment for V. To compute S, find the most general unifier of E_1 over V with a subexpression of the expression being reduced.

Notion 30: Equality of Expressions

When we look at the two arabic[13] numerals 1,935 and 51,837, we rapidly become aware of whether they represent the same number. This process is often seen as so natural as to not require further explanation. However, if asked about the equality of two roman numerals, such as `CCCCC` and `CCCCLXXXXVIIIII`, the situation is not so clear. Nevertheless, by computation, `IIIII = V`, `VV = X`, `XXXXX = L`, `LL = C`, we can work out in this case that they are equal. Returning to the arabic example, we see that there is a computation. Each symbol of one is checked against the other, to see whether they are all equal.

The form of arabic numerals is canonical. Symbol-by-symbol equality testing of two numerals is a correct test for equality of the numbers. This is because we do insist that arabic numerals are normalised, 'leventy 'leven is 121, but does not normally obtain a high score in an arithmetic exam.

We usually accept non-normalised roman numerals, so we need to do more work. But if we insist that in a roman numeral any symbol that can be replaced by a higher value symbol must be, and that the larger-valued symbols are placed to the left, then we are using a canonical form for the roman numerals, and a symbol-by-symbol check works.

Generically the equality of two tuples is component by component. A set is more troublesome. When we write down a set we give an order, {a,b} is distinct on the page from {b,a} and there is nothing we can do about this. So, we have to learn that the order makes no difference. Proving equality of sets can be quite difficult.

As for rational numbers, when are they equal? Is 4/6 the same as 2/3? The generic test for equality of a/b and c/d is $ad = bc$. But there is a canonical form of rational numbers, no common factors. We can determine it by dividing out any common factors: 4/6 becomes 2/3, which is symbol-by-symbol equal to 2/3.

Strictly, symbol-by-symbol equality is not simple. For you to identify that this x is the same as this x, you have to do a complex vision-

[13] Strictly, this is Moroccan Arabs, the rest use the Urdu system.

processing computation, in your head. Further under the right conditions you would say that this x is the same as this `x` even though they are in different fonts. Writing a program that determines the equality of handwritten symbols on a page is a non-trivial exercise.

What, then, is equality?

By now you should have some doubt that there is such a concept. I hope, at least, I have demonstrated that there is no *singular natural* equality. Rather, our idea of when two things are equal is defined for the given context. For each data type we construct we must decide what equality will mean in that context.

What is equality in computation?

It is a binary relation, a predicate, a Boolean valued function. But it must satisfy some conditions. Firstly, it is clear that $A = A$; nothing that we would call equality could fail in this condition. Then $A = B$ means that $B = A$. This excludes relations such as $A > B$, which behave a little like equality, but are not any type of equality. Also, and very commonly, we reason $A = B = C$ so $A = C$; again this is very deeply fundamental to our concept of equality.

The computer provides us with a definition of byte equality. A memory block is a sequence of bytes, and can be tested for equality as such, using component by component equality. In C, this is automatically available for structs defined by the programmer. Often byte-wise equality of a memory block is sufficient to imply equality of the two abstract data elements. But, as for rational numbers, it is often not necessary. Thus, byte-wise and abstract equality may be different. For this reason,[14] Java does not allow byte-wise equality. You have to define what you mean by equality when you define Java classes, or you cannot test for equality.

It is also logicaly possible (even if inadvisable) for the meaning of a data element to depend on its location in memory, thus making byte-wise equality insufficient for determining abstract equality.

[14] And others such as some security reasons.

Notion 31: Equational Reasoning

Equality (see page 66) is well defined by its nature.

$A = A$,
$A = B \Rightarrow B = A$,
$A = B = C \Rightarrow A = C$,

That is, equality is a reflexive, symmetric, transitive binary operation. However, in reasoning with equality it is often assumed without explicit statement that substitution respects equality. But, in C, given `x+y=y+x`, and substituting `z++` for `x`, we get `(z++)+y == y+(z++)`. The exact status of this equality has varied over the years. Some compilers evaluated `++` as it occurred; others waited until after the expression; others did whatever they pleased at the time. The logical status of this equality is contingent.

We need to include an explicit axiom into our logic system.

$A = B \Rightarrow m(A) = m(B)$

If we know $A = B$, then for any(see note 10) expression m we know that $m(A) = m(B)$, even when we do not know what either side of the equality means. A system that satisfied these axioms is said to admit equational reasoning.

Substitution of equals for equals in pure expressions preserves equality because there are no side effects. But in many languages, expressions have side effects. The value of the expression has been taken to be the return value within the language. This is very misleading. For example, given $x + y = y + x$, we might substitute values to obtain $f(1) + y = y + f(2)$, which is not generically true. But, you say, we substituted a different value of the argument to f in each case, this is why the equality is violated. But this is exactly what side effects are about. Hidden parameters and hidden return values.

Systems with side effects do not admit equational reasoning if the side effects are ignored in the equality of the expressions. An expression with side effects can be turned into one without by just listing the variables

affected on the side explicitly in the argument lists.

Given, `int f(int x){y=x; return z;}`, the call `a=f(b)` has side effects, but if f is defined by $f(x, z) = (z, y)$, then we have the same effect using $(a, y) = f(b, z)$, and all expressions are pure.

The problem with equational reasoning in languages with side effects is just that it is common to ignore part of the action of the expression. This is essentially a misinterpretation of the notion of equality, rather than a breakdown of equational reasoning.

However, what is true is that certain languages make it very difficult to determine when two expressions are identical. A language like Haskell that admits equational reasoning does so by requiring that all effects be declared.

The equality $x + y = y + x$ is based on the idea that x and y are numbers (for example), but `i++` is not a number, but rather a piece of code. Suppose that `i==2`, can we substitute 2 for `i++`? No, even though this is its value, since `i++` is actually `{t=i; i=i+1; return t;}`, which is not the same as the number 2, or even the variable `i`.

It is difficult to avoid side effects when dealing with interaction. Printing a document is, in itself, a side effect. The essential way around this is to express the program as a whole as a function from a string of input events to a string of output events.

Another complication is that if the hidden parameters of a function are changed it is not required to change the code in which it is used. But if all have to be declared, then the code must be changed every where the function is used ... resulting in non-modular code. The generic way out of this is to pass records, rather than inline tuple lists; and then the extra fields in the records do not have to be explicitly mentioned in the code. But a full solution still involves more thinking.

The fact remains, however, equational reasoning is often worth the effort due to the increase in robustness of code.

Notion 32: Unification Reduction

Unification reduction is the arithmetic of computation. It is at once a low-level mechanism and a high-level concept. It takes an expression as representing nothing other than itself. It is a casebook of those things you were told not to do in high school (see note 2) like identifying number and numeral (see page 58).

Let E_1 and E_2 be two expressions using variable symbols $x_1..x_n$. To *unify* (see page 64) E_1 and E_2 is to find expressions to subsititute for $x_1..x_n$ so that E_1 and E_2 become the same expression. If we substitute $x = 1$ and $y = 2$ in "$x + 2$" and "$1 + y$", we get in each case "$1 + 2$".

Unification commonly occurs in standard programming languages in the evaluation of function calls. We are told that $f(x) = x^2$. We unify $f(a+b)$ with $f(x)$ to get $x = a+b$, and then substitute into x^2 to obtain $f(a + b) = (a + b)^2$. The sudden appearance of a pair of parentheses is important and their absence is a common novice error, but this is simply the original implicit grouping being made explict.

We may need multiple substitutions going both ways. For example (x, z) is unified with $(1, (1, y))$ by the substitution $\{x = 1, y = 2, z = (1, 2)\}$, or more generally by $\{x = 1, y = a, z = (1, a)\}$, where a is a free parameter.

The process of changing one expression into another by substituting the values for variable symbols is *reduction*. The full process involves three exressions, E_1, E_2, and E_3. We unify E_1 and E_2 to obtain values for variable symbols, and then substitute these values into E_3. Though often mentioned only implicitly, and accepted without comment, unification reduction is deeply fundamental to the concept of both computation and proof.

Languages such as C and Java use limited unification reduction in function calls. In compiling the expression `f(x*x,y+2)`, the identification of expressions to substitute for the formal parameters of `f` must be made. But this is only unification of flat tuples. In Haskell, we can unify `f(x,(a,b),[c,d,e])` or any other compound structure with the original declaration of `f`. But, no repetition of variable symbols is allowed in the formal parameters. In Haskell, we cannot define equality with the

clause $eq(x, x)$. Prolog does allow this. In Prolog, full pure unification (with no axioms) is used for procedure calls. So, a problem of the form $f(x)$ may be changed to $sqr(add(x, 1))$, or $sqr(add(x, 1))$ to $f(x)$.

The complete state of a countable state machine may be described as a string of characters such as `"(1,23,a,3.14)"` listing the values of all the variables as they would be printed. Syntax respecting subsitutions on such strings provide a mechanism for changing the state of the machine. Any computable process can be expressed in this manner.

An axiom is a logical starting point, the definition of what we may assume. Typically axioms are expressed as unification reductions, even if this point is not formally recognised. The factorial function may be defined by $0! = 1$ and $n! = n \times (n - 1)!$. The 1st axiom states that we may replace the subexpression `"0!"` by `"1"`, and the 2nd axiom is a non-trivial unification. Repeated application of this unification reduction will change `3!` into `3×(3-1)×(3-1-1)×(3-1-1-1)!`. With only the given axioms this would continue indefinitely. But, if we have arithmetic defined by unification reduction (see page 58), we can change it into `3×2×1×0!`, and thence into `3×2×1×1` and `6`.

More general systems of unification with a variety of axioms are studied in universal algebra (see note 3), a study of which is highly recommended for any serious programmer. The semantics of code, including imperative, procedural code, can be expressed as unification reduction. (see page 74)

In a reduction all meaning is erased from the computation. Separation of syntax from semantics is the universal task of the programmer. Meaning is a human-imposed association. A formal system (a program) is a pure syntactic system that is homomorphic to the human system with meaning.

Many bugs arise from the incorrect assumption that the program is imbued with natural meaning. The truth is that the meaning is only in the head of the programmer. Pushing further, we find that the meaning of meaning is emphemeral, and most likely only exists in a single head, and cannot be transfered even from human to human. We transmit only the syntax of written or spoken word.

Exercise 8: Unification Reduction Engine

Write a simple unification-reduction engine.

Of course this exercise is not so simple. But it should be very illuminating for those that attempt it. The basic operation is that the engine matches an expression $E(x, y)$ against a the left-hand side of the substitutions in a database of rules $(A_1(x, y) \rightarrow B_1(x, y), \ldots)$ best thought of as a tuple rather than a set. Finding one that matches, we then substitute the values into the right-hand side to obtain the next expression. For example, with $Add(F(x), y))$, and having the rule $Add(x, y) = Add(y, x)$, reduce the expression to $Add(y, F(x))$. This is repeated until no reductions are possible.

Parsers

Although the target of the exercise is unification and not parsing, attempting to avoid all parsing will mean constant hand built data structures and recompilation. So it is best that your program can parse basic expressions from the keyboard, or from a file. Possibly the easiest expressions to parse are s-expressions. The expression `(f a b c)` means f applied to (a, b, c). All brackets must be included, so that $f(x + y)$ becomes `(f (+ x y))`. However, anyone who wishes to build more complex expression parsing is encouraged to do so.

Internally, an expression should be stored and manipulated as a tree. In fact, an s-expression can be seen simply as a notation for a tree. For example, `(f a b)` means, *the tree with root weighting* `f`*, and subtrees* `a` *and* `b`.

Expressions

Trees are fairly easy to implement in C, Java, Prolog or Haskell. In C, a `struct node {symbol r; node *child[10]};` will do even though it limits the number of arguments to 10. For the more enthusiastic, a linked list of arguments `struct node {symbol r; node *child; node *next};` is more general. In Java, just use analogous classes. You cannot use lists of lists in Haskell to represent an s-expression because the type

system will object to the infinite signature. However, it is possible to define an analgous new data structure that does not suffer from this limitation. Prolog comes with unification reduction inbuilt; however, getting at it for the purposes of retrieving the required environment still needs some effort.

Unification

The simplest case treats all symbols as variable, get this running first. Later look at including the list of constants that are not to be given values. The giving of a value to a constant can be checked on the fly or as post-processing on the all-variable case.

We can replace all occurrences of a variable in values at the time that its value becomes apparent. Then it is simple and fast to check for cyclic dependencies. But it is logically neater to eliminate cycles afterwards, and the program could instead give the minimum set of axioms under which the expressions unify.

We need a mechanism to indication non-unification. We cannot use an empty set for this, since two expressions might unify with an empty environment. This suggests adding a special element, not-an-environment. Handling this with an exception in Java is also suitable but does not necessarly constitute improved code.

Substitution

Substitution of values for variable symbols is fairly simple recursively. For example, this is a simple routine to subsitute values into a Prolog List. The arguemnts are the environment (a list of pairs), the original list, and the final list with the variables replaced by their values.

```
subst([],X,X) :- !.
subst(L,[A|B],[X|Y]) :- subst(L,A,X),subst(L,B,Y),!.
subst([(A,B)|_],A,B) :- !.
subst([_|B],X,Z) :- subst(B,X,Z), !.
```

Notion 33: Code Reduction

An expression is a correctly bracketed (see note 13) sequence of symbols that may be written down on a piece of paper or stored in an electronic digital computer memory. Sometimes the bracketing is irrelevant, as in $1 + 2 + 3$, where it is left out because it makes no difference to the value, or implicit, as in $2 + 3 \times 4$, where there is a convention that the terms are bracketed thusly, $2 + (3 \times 4)$. A subexpression is a part of an expression which respects the (possibly implicit) bracketing. So $2 + 3$ is not a subexpression of $2 + 3 \times 4$ because it cuts across the implicit brackets. Given a reduction, $x * y \rightarrow y * x$, where x and y are unification variables, we can apply it to subexpression in situ, for example, as used to produce the reduction $3 + (5 * 6) \rightarrow 3 + (6 * 5)$. This concept is the most primitive in any discussion of digital computation as well as in algebra. Without this or a similar concept we can do no computation, with this we can do all possible computations.

We have a reduction process with suggested inuitive meanings:

`(add x 0)`	$\rightarrow$	`x`	$x + 0$	$=$	x
`(add x (s y))`	$\rightarrow$	`(s (add x y))`	$x + (y + 1)$	$=$	$(x + y) + 1$
`(mul x 0)`	$\rightarrow$	`0`	$x * 0$	$=$	0
`(mul x (s y))`	$\rightarrow$	`(add x (mul x y))`	$x * (y + 1)$	$=$	$(x * y) + x$

This defines addition and multiplication of non-negative integers,

for example,

```
    (mul (s(s 0)) (s(s 0)))
→   (add (s(s 0)) (mul (s(s 0)) (s 0)))
→   (add (s(s 0)) (add (s(s 0)) (mul (s(s 0)) 0)))
→   (add (s(s 0)) (add (s(s 0)) 0))
→   (add (s(s 0)) (s(s 0)))
→   (s (add (s(s 0)) (s 0)))
→   (s (s (add (s(s 0)) 0)))
→   (s (s (s (s 0))))
```

The expression being reduced is an s-expression, and is valid Scheme code. The above is an example of evaluation of Scheme by expression reduction.

When we write a pure function in C code we can think of it in exactly the same way:

```
int f(int x){return x ? x*f(x-1) : 1;}
```

may be read as

```
f(x) → (x ? x*f(x-1) : 1)
```

so

```
    f(4)
→   4 ? 4*f(3) : 1
→   4*f(3)
→   4*(3 ? 3*f(2) : 1)
→   4*(3*f(2))
→   4*(3*(2 ? 2*f(1) : 1))
→   4*(3*(2*f(1)))
→   4*(3*(2*(1 ? 1*f(0) : 1))))
→   4*(3*(2*(1*f(0))))
→   4*(3*(2*(1*(0?0*f(-1):1)))))
→   4*(3*(2*(1*1))))
→   4*(3*(2*1)))
→   4*(3*2)
→   4*6
→   24
```

Pure unification string productions of this type can be used to define the semantics of a language precisely at a reasonably high-level. It avoids the complications of trying to explain how the code will be executed on a Von Neumann machine. There are ways to translate this type of definition into a compiler for the language. However, this is a very powerful method, and it is easy to define semantics that can bog down the largest computer. Some restraint is required when using it to define a language.

Notion 34: Programming With Logic

Predicate calculus can be used as a programming language. The principle is straightforward. For simplicity we look only at pure programs that read data in and write out a result.

Take, as an example, the computation of the greatest common divisor, $gcd(x, y)$, which satisfies the axioms on the right using the convention that a variable is universally qualified by default.

$$\begin{array}{lcl} gcd(a, a) & = & a \\ gcd(x, y) & = & gcd(y, x) \\ gcd(x, y) & = & gcd(x, y - x) \end{array}$$

We already have enough information to determine $gcd(50, 15)$,

$$\begin{array}{rl} & gcd(50, 15) \\ = & gcd(15, 50) \\ = & gcd(15, 35) \\ = & gcd(15, 20) \\ = & gcd(5, 15) \\ = & gcd(5, 10) \\ = & gcd(5, 5) \\ = & 5 \end{array}$$

Each step is a unification reduction (see page 70). By trying all possible rules at each step, and producing a tree of resulting expressions, an automated process can discover a node in which the term `gcd` does not exist. This is a leaf node on the tree, and is the required answer.

Pure logic programming is an axiom-building exercise. We presume that the computer provides the ability to determine the logical conclusions. This is no different from the requirement that the computer provide the ability to follow through the instructions given to it in C, Java, or Haskell.

Real logic programming is not programming in the sense the term is usually used, it is too powerful. It removes all the basic issues of programming, and replaces them by only the problem of writing specifications, and working out if they satisfy human desires. But real logic programming is not possible, since the problem as posed is non-computable.

Pragmatic logic programming involves a lot of work in determining which of several logically equivalent axiom sets will reduce efficiently under a given unification reduction scheme, as well as including meta-logical information such as advice on which order the unifcations should be tried.

Many aspects of C and Java are logical in nature. A declaration `int x;` is an assertion that `x` is in the `int` set of data elements. A function definition `f(x){int y = x+x; return 2*y;}` asserts an equality between evaluation of $f(x)$ and evaluation of the body of the code.

In Haskell, a logic style program is possible:

```
fact 0 = 1
fact n = n*(fact(n-1)).
```

These Haskell clauses look and act very much like their pure logic interpretation, indeed even the axiom `fact (n+1) = (n+1)*(fact n)` is valid Haskell.

It is trivial to interpret an appropriately written Haskell program as a pure logical scheme. What makes Haskell not a logic programming language is that it does not go in reverse. Logically, the assertion `fact n (fact(n+1))/(n+1)` is just as valid. But working from top to bottom and left to right, Haskell would not finish the computation.

In principle, the notion of *programming in logic* refers not to the construction of a program as a set of predicate calculus axioms, but rather the requirement that the behaviour of the program be unchanged by permuting the axioms in the program, or by reversing the logic of an individual axiom. A logic program is an unordered set of bi-directional equality assertions.

In practice, the art of logic programming is *all about* choosing the order of assertions and the direction of equalities, to make the computations work in practical time. There are no real logic languages, but much progress has been made in the latter part of the 20th century, largely under the guise of optimising compilers and compilers for functional languages.

Notion 35: Negation in Logic Programming

Negation is a complex and subtle concept.

If I say I am lying, then am I lying?

I define the negation of a predicate A, to be the predicate (not A) such that ((not A) xor A) is always true. But it is not axiomatic of all logic systems that such a predicate exists, or that if it exists it will be unique.

The negation of *Fred has a mouse in his pocket* might be *Fred does not have a mouse in his pocket.* But what is the negation of *The present king of France has a mouse in his pocket.* If we say that the present king of France has no mice, we assume that the king exists. There is no problem talking of fictional characters, such as Sherlock Holmes (who has no mice). But, we are talking of a real king of France. The negation is closer to *the king is mouseless, or there is no king.*

Stating the original with more caution, we say *someone is Fred, and something is a mouse, and that something is in the someone's pocket.*

Exists x : Exists y : Fred(x) and mouse(y) and inPocket(x,y)

Negating this

Forall x : Forall y : not Fred(x) or not mouse(y) or not inPocket(x,y)

Either Fred does not exist, or the mouse does not exist, or the mouse is not in Fred's pocket. This very close to *no one called Fred has a mouse in his pocket.* But I did not mean *just anyone* called Fred; I have a specific person in mind. So, I find myself asserting that this specific real individual might not exist.

Is it logically valid to say that Fred (my boss from when I worked as a control systems engineer) does not exist? If not, then from whom did I just receive an email? Is existence a property of an object? I do not claim this conundrum is a logical paradox. There are plausible responses. But our choice affects the method for negating, and whether negation can be done at all.

Negation in logic programming is similarly fraught with difficulty.

We might define negation by automated logical rules for transforming a predicate. But we would have difficulty knowing if these rules are correct, consistent, or complete. It might be better in practice to require the programmer to supply the meaning of negation of each predicate as needed. This is similar to requiring the programmer, as Java does, to supply the definition of equality for a new datatype.

Alternatively, as in Prolog, we may define negation as failure. If a search fails to find a goal, then the negation of the goal is asserted true. Thus, `(not g)` means run the proof technique, see if it proves `g`; if not, then take this as a proof of `(not g)`. Mind you, how then are we to list all the solution space of `(not g(X))`? If `g(X)` results in an infinite stream of integral instances, then we cannot be sure to compute whether any specific integer satisfies `(not g)`.

Failure to prove and proof of failure are two different things. Given only that X is a real number we would fail to prove that it is positive, and fail to prove it is negative, and thus we have proved that an arbitrary real number is zero.

The logically correct Prolog program

```
positive(1).
negative(-1).
neutral(X) :- not(positive(X)), not(negative(X)).
```

returns the following rather uniluminating results:

```
neutral(2) == yes.
neutral(X) == no.
```

Thus, Prolog has a negation but it should be called *not in the database*. Prolog negation does not have the properties normally required of logical negation, if you want a negation, then you should decide for yourself, and explicitly code, what you had in mind, case by case. The use of the metalogical operators is advised, but great caution is needed to avoid introducing other illogical results.

Notion 36: Impure Lambda Calculus

The phrase *lambda calculus* means different things to different people.[15] The common theme is computation with function-like elements. In this section, we motivate the lambda calculus as a whole by looking at some technical issues of the use of functions.

In orthodox languages we do not need to name an integer to use it. To pass the square of the integer 123 to a function `f`, we use an expression such as `f(123*123)`. We do not need to define a variable `int x = 123*123;` and then call `f(x)`. But, if we wish to pass the double application of a function `f` to a numerical integration routine `integrate` we need to declare `g(x)=f(f(x))` and then pass `g` as in `integrate(a,b,g)`.

This restriction on function declaration is sometimes conflated with the notion of functions not being first-class datatype. But the notion of first-class datatype is more typically used to mean that the datatype can be stored in arrays, passed as arguments, and returned as values from functions. By this definition, function is a first-class datatype in C, since (implicit) function pointers provided the essentials of this behaviour. But in C we are unable to construct *new* functions without some form of machine-level hacking. It is the ability to construct and operate on arbitrary elements of the datatype that make all the difference in practice between a datatype that is clumsy or one that is deft.

The first thing that lambda calculus gives is a mechanism for in-place function construction and use. For example, $(\lambda x \cdot x^2)$ is the function that squares its argument. More generally $(\lambda x_1..x_n \cdot E(x_1,..,x_n))$ is a function that takes n arguments and returns the value of the indicated expression. In this context, the lambda calculus appears to be a simple syntactic convenience, in which the arguments and code for an anonymous function are packaged into an expression that can be used at the place of the call.

Our earlier example might become `integrate(lambda(x){f(f(x)})`. Notice that the expression `f(f(x))` still occurs because this is our way of expressing the required function. The point is that now we do not

[15] This state of affairs is not unprecedented: see, for example, the large variety of approaches to the meaning of the term *differential calculus*.

need to make a declaration, which in particular means that we do not need to know when we write our code what functions we are going to use. Imagine the problems caused if you had to declare all the integers (1,2,3,45 ...) that were going to appear during the running of a program.

Notice that in the above piece of code a pair of braces {} appears inside parentheses (). In practice, this is often a sign that something unusual is occurring. It occurs in Java when anonymous extension classes are used; anonymous extension classes can be used in a manner very similar to a scheme lambda expression. While at first seeming a small detail, syntactic sugar, it is the ability to use a class anonymously that makes it easily adaptable. In principle, C++ can duplicate this effect by the use of explicit class definitions, but it becomes tedious, and moves the definition of the function away from the place it is used.

Lambda expressions could be used to good effect in other areas of study. For example, in algebra the expression x^2 is often confused with the function $(\lambda x \rightarrow x^2)$. Taking the expression x^2 and producing $(\lambda x \rightarrow x^2)$ is called *lambda abstraction*.

The Newtonian approach to differentiation $f'(x)$ requires us to predefine f, while the Leibnitzian $\frac{d}{dx}x^2$, allows an implicit lambda abstraction. It is interesting to note that Leibnitz was much more strongly into constructive logic and computer science than was Newton. However, beware that $\frac{d}{dx}x^2 = 2x$ uses implicit lambda abstraction. Perhaps we should say, $d(\lambda x.x^2) = (\lambda x.2x)$, just to be careful.

Partial differentiation could be handled by taking an expression, with no λs, and abstracting it. That is ...

$$\frac{\partial}{\partial x}E = (D\ (\lambda x.E))\ x$$

where D is an operator that takes a function of a single variable and returns the derivative. This is a formal, constructive, definition for the intuition that a partial derivative treats all other variables as constants.

Notion 37: Pure Lambda Calculus

Normally unless, and even perhaps if, you are heavily into formal logics, it is best to be introduced to the impure lambda calculus (see page 80) for motivation before tackling the purest form described here. It might also help to review substitution computation (see page 52).

Given an alphabet Σ (intuitively the collection of variable names), not containing "(", ")", "." or "λ", we define the set of lambda expressions $\Lambda(\Sigma)$ over Σ, inductively, as the smallest (see note 18) set that satisfies the following three axioms:

$x \in \Sigma \Rightarrow x \in \Lambda(\Sigma)$ ..... we need to start somewhere
$A, B \in \Lambda(\Sigma) \Rightarrow (AB) \in \Lambda(\Sigma)$ ..... think *function application*
$x \in \Sigma, E \in \Lambda(\Sigma) \Rightarrow (\lambda x.E) \in \Lambda(\Sigma)$ ..... think *function construction*

In $(\lambda p.B)$, p is the *parameter* and B the *body*. The substitution $[x \to X]$, of the value X for the variable name x, must respect the declaration of parameters in sub expressions that mask the body of the subexpressions from the scope of the variable x.

$$\begin{array}{lcl} [x \to X]x & = & X \\ [x \to X](AB) & = & (([x \to X]A)([x \to X]B)) \\ [x \to X](\lambda x.E) & = & (\lambda x.E) \\ [x \to X](\lambda y.E) & = & (\lambda y.[x \to X]E) \end{array}$$

There are three reductions, alpha, beta and gamma:

Alpha: $(\lambda x.E) \to (\lambda y.[x \to y]E)$ ..... change of parameter
Beta: $(\lambda x.E)B \to [x \to B]E$ ..... function application
Gamma: $[x \to B]E \to (\lambda x.E)B$ ..... reverse application

Alpha reduction as stated above is not always valid. For example, $(\lambda y.[x \to y]xy)$ is $(\lambda y.yy)$. Alpha reduction $[x \to y]$ on $(\lambda x.E)$ is valid exactly when y does not occur within the scope of x in E. A free occurrence is not in the scope of any appropriate parameter declaration. There are complete syntactic rules for *free* and *scope*. But, pragmatically, alpha reduction is valid exactly when $[y \to x][x \to y]E = E$, that is, when the substitution can be reversed.

We have a collection of expressions equipped with a reduction system. This is a *directed* graph. But each alpha reduction can be reversed by another alpha reduction, and each beta reduction has its inverse in the gamma reductions. Thus, each link can be traversed both ways. Expression A reduces to B exactly when B reduces to A. The reduction graph is undirected. Also, it is clear that if A reduces to B, which reduces to C, A reduces to C, by catenating the reductions from A to B and B to C. We include as a special case the trivial reduction of A to A.

Thus, reduction is reflexive, symmetric, and transitive, and defines a form of equivalence (see page 68) between lambda expressions. Two expressions are equivalent if they are in the same connected component of the reduction graph.

A *normal* lambda expression is one which admits no beta reductions. All normal forms equivalent to a given lambda expression are alpha equivalent. That is, if you can reduce lambda A to two distinct primitives p and q, then p and q are identical except for a change of the names of the parameters. We may think of p as the *value* of A, while A is a compound expression whose value is p.

Each beta reduction is associated with a λ in the expression. If we repeatedly select the outermost (leftmost) λ that can be reduced and reduce it, then if a normal form exists it will be reached after a finite number of reductions. Speaking informally, any lambda for factorial must satisfy $f = (\lambda\ n\ .\ n = 0\ ?\ 1\ :\ (n * (f\ (n - 1))))$. So, factorial has no normal form, and neither does factorial of -1, but factorial of 3 does.

The lambda calculus is one of the more successful of a variety of systems for formalising the notion of computation. It has been suggested at times as the basis for physical computational devices, but our desktop computers are based more closely on the Turing and Von Neumann machines. The physical construction of a lambda engine requires large, fast stacks. Possible, but perhaps more complicated than current computers. In its favour, it has been argued that the increased ability to prove lambda systems correct would offset any difficulties. But the commercial reality is that we are less likely to see lambda machines on every desk than electric cars in every garage.

Notion 38: Pure Lambda Arithmetic

Impure lambda calculus (see page 80) is additional syntax for defining functions in terms of operators that we already have. The pure lambda calculus, taken strictly, has no other operators, only lambda expressions. This includes no numbers, no arithmetic, and no conditional statements. The meaning of a pure lambda expression is imposed by its use, outside the context of the calculus itself. Here we discuss a few aspects of how to program in pure lambda calculus. The principle is to construct lambda expressions that *behave like* the structure we are trying to program. This is true of programming in languages such as C, Java or Scheme, but this point is much more explicit in pure lambda calculus.

To construct the positive integers we need a collection of expressions that encode the concept of *that many*. For example, $1 \equiv (\lambda x.x)$, $2 \equiv (\lambda x.(xx))$, $3 \equiv (\lambda x.((xx)x))$ etc. For simplicity we assume left association in the absence of brackets, and obtain $n \equiv (\lambda x.xxxx..x)$ with n occurrences of x.

Although it is then apparent to the programmer what needs to be done to operate on these numbers, pure lambda calculus is heavily flavoured by the point that it is *impossible* for one lambda expression to determine the actual construction of another. Only the external behaviour is available, and that only through explicit test cases.

To justify its meaning as the integers, within lambda calculus we must define increment as a lambda expression. To simplify the expressions, we take the convention of curried expressions in which $(\lambda xy.E) \equiv (\lambda x.(\lambda y.E))$. Like the left associative brackets, this is purely a shorthand, and not the introduction of a different type of expression. It is then possible to define $inc \equiv (\lambda nx.nxx)$.

For a non-trivial implementation of the integers we would want addition. Reviewing the previous construction we might be tempted to try $(\lambda nmx.(nx)(mx))$, certainly this gets the right number of occurrences of x, but they are bracketed in the wrong manner.

The problem is that we need to construct $((((xx)x)x)x)$ from $((xx)x)$ and (xx), this means putting the (xx) in the place of the first x in

$((xx)x)$, which we cannot get at. The way out of this conundrum is to put this facility into the number from the beginning. Redefine n to be $(\lambda yx.yxxx..x)$ with $n-1$ occurences of x. Thus, $inc = (\lambda nyx.nyxx)$, and now $add = (\lambda nmyx.m(nyx)x)$.

Now that we have this bit of inner workings to use, we can look to multiply as repeated addition, except that we need n applications of $+m$, to get $n \times m$. If we reverse the application order so that $n = (\lambda xy.(x(x(x \ldots (xy) \ldots))$, then inc and add work roughly as above, and $mul = (\lambda nmxy.m(add\ n)(\lambda xy.y))$. This amounts to adding n to 0 a total of m times.

The representation we have developed is reasonably servicable. The definition of the number n as an operator that applies a function n times means that many arithmetic operations are easy to define. Powers, for example could be defined by repeated multiplication. Equality to zero can be tested by a slightly more subtle approach. The function $(\lambda x.\texttt{false})$ applied zero times to `true` is `true`, any more times, and it is `false`. However, we have not yet discussed how to build conditionals.

We still do not have decrement. One problem with decrement is decrement of 0, which is an error, but we have no mechanism for handling it. By default typically a non-numerical lambda expression will result. But the larger problem is that further operations are not produced in any obvious manner, because the number does not already have the required facility.

An (more efficient) alternative is to represent the number in binary, for example, $(\lambda xy.xyxxyyxy)$. To build the operations, we need conditional structures, which are discussed elsewhere (see page 86).

The reason for the mass of technicalities involved in developing arithmetic in pure lambda calculus is that we are insisting on doing everything from the beginning. This is using pure lambda calculus as a form of assembly language. Once all these pieces are defined, we can proceed at a higher level.

Notion 39: Pure Lambda Flow Control

The conditional can be constructed in pure lambda calculus by an almost trivial mechanism once we realise the manner in which the conditional is used. Refering to the C language, `(a ? b : c)` returns either `b` or `c` depending on whether `a` is true or false. That is, `a` is a mechanism for deciding which of `b` or `c` to return. Let true and false represent the two options. Now, we can define this directly in lambda. $true = (\lambda xy.x)$ and $false = (\lambda xy.y)$. Then if A is an expression that returns true or false, we have $if = (\lambda abc.abc)$.

This gives the conditional, but how do we make operators that return the appropriate truth value? One idea is to build them into the datatype. So a number n can be $(\lambda t.tab)$, where a is the answer to the question *are you zero*, and b is the number as represented by some other mechanism (see page 84). This makes the process extremely close in style to object programming. We cannot look inside the datatype we have been given, but it will listen to certain requests for information, mediated by methods that it defines, and we can use.

Although the convention of using a name in place of a lambda expression is certainly used heavily, we must recall that this is shorthand for writing out the expression. It is easy to fall into the error of using a name in its own definition. It is an error because if we do then it is no longer pure lambda calculus, but an extension allowing recursive definition.

The Y-combinator, however, is a mechanism which provides the practical power of recursive definition without actually being recursive itself (see page 88). In essence, given a function defined recursively by $f(x) = E(f, x)$, the Y-combinator has the property that $YE = f$; Y manages to find the function that would have been defined by the recursive defintion using E.

Now, we can define (see page 238) the while-loop

```
while t s x = if t x then x else while t s (s x)
```

Exercise 9: Lambda Reduction

It is possible to write a simple lazy lambda reduction engine in Java (see page 262). This or a similar lambda engine can be used to test and debug this exercise. The Java code can get tangled: if you are happy with parsing text files you might want to consider a lambda to Java converter, using lambda calculus syntax such as `(L x . x * x)` for lambda expressions.

1. Implement natural numbers with addition.
2. Include subtraction.
3. Include multiplication.
4. Include the ability to test a number for being zero.
5. Include a numeric error value for an invalid subtraction.
6. Include the ability to test any number to see if it is valid.
7. Include a test for equality that subtracts and tests for zero.
8. Include test of relative size that subtracts and tests for validity.
9. Implement factorial using pure lambda all the way.
10. Implement fibonacci using pure lambda all the way.

There are a number of approaches. In the minimumalist approach, with only increment, decrement, and a zero test, we can define the rest by

```
add(n,m) = add(n-1,m+1)
sub(n,m) = sub(n-1,m-1)
mul(n,m) = add(n,mul(n,m-1))
```

But there are more efficient mechanisms. Initially you may find it easier to assume that the subtraction is valid, rather than including the error code up front.

Notion 1: Computing Hyperfactorial Values

We define the Y-combinator (see page 86), also known as the fixed point operator, to be the lambda expression:

$Y = (\lambda f x.(f(xx)))(\lambda f x.(f(xx)))$

Clearly, using beta reduction, $Yf = f(Yf)$. So Yf is a fixed point of f, regardless of what f is. Thus, Y finds fixed points. A function may have more than one fixed point. Y finds the simplest fixed point (in a strict technical sense that we will not define here).

A recursive definition such as

```
fact x = if x==0 then 1 else x * (fact(x-1))
```

can be abstracted to

```
hfact fact x = if x==0 then 1 else x * (fact(x-1))
```

The original factorial function is then the minimal fixed point of `hfact`, which is then extracted by the Y-combinator. Detailed understanding, however, is not provided by this definition. But lack of understanding does not prevent the use of the definition to explore its meaning. You do not know what the function $f(x) = x^{96} + 5x^{23} + 45x$ looks like, but you can compute some values, such as $f(2)$. It is similar with lambda expressions.

As $sqr(x) = x * x$ leads us to $sqr(6) = 6 * 6 = 36$, so does

$\text{hf}(f) = \lambda x.(x == 0?1 : x * f(x-1))$ lead us to

$\text{hf}(\lambda x.0) = \lambda x.(x == 0?1 : x * (\lambda x.0)(x-1))$

noting that $(\lambda x.0)(x-1) = 0$ we have:

$\text{hf}(\lambda x.0) = \lambda x.(x == 0?1 : x * 0) = \lambda x.(x == 0?1 : 0)$

As $sqr^2(6) = sqr(sqr(6)) = sqr(36) = 1,295$ we look at $\text{hf}^2(\lambda x.0)$.

$\text{hf}^2(\lambda x.0) = \text{hf}(\text{hf}(\lambda x.0)) = \text{hf}(\lambda x.(x == 0?1 : 0))$

By straight substitution we get

$= \lambda x.(x == 0?1 : x * (\lambda x.(x == 0?1 : 0))(x - 1))$

Applying $\lambda x.(x == 0?1 : 0))$ to the argument $x - 1$ we get:

$= \lambda x.(x == 0?1 : x * (x - 1 == 0?1 : 0))$

Noting that $x - 1 == 0$ is the same as $x == 1$, we get

$= \lambda x.(x == 0?1 : x * (x == 1?1 : 0))$

Distributing $*$ over ? :, we get

$= \lambda x.(x == 0?1 : (x == 1?x : 0))$

Finally, the true condition is only evaluated when $x == 1$, so

$= \lambda x.(x == 0?1 : (x == 1?1 : 0))$

Repeating these steps we find that

$\text{hf}^3(\lambda x.0) = \lambda x.(x == 0?1 : (x == 1?1 : (x == 2?2 : 0)))$

Continuing $\text{hf}^n(\lambda x.0)$ expands out the first $n - 1$ values of factorial.

In particular, $(hf^{n+1}(\lambda x.0))(n) = n!$

We can now see the mechanism by which the Y-combinator works. Given an expression $E(f)$ in f, which might be used to define f recursively as in $f(x) = E(f)(x)$, the Y-combinator simply applies $E(f)$ to x without ever asking what f actually is. If $f(x) = E(f)(x)$ is a legitimate recursive defition of f, suitable for use in a program, then in the repeated application of $E(f)$ to x eventually it will turn out that the value of f is not required. Thus, the Y combinator manages to return the required value.

Notion 40: S-K Combinators

One fine day longer ago than I care to admit, I noticed on the wall of the corridor of the university I was attending a circular about a talk on programming. *I will show you* it said *how to write a program with no recursion, no iteration, and no variables.* I was fascinated. I had by then realised that recursion and iteration where essentially the same concept. You could build a language with either iteration or recursion playing the central role of allowing repeated application of some piece of code to some data stored in some variable. But to lose both? How could you program anything?

Some reflection on the matter formed in my mind a concept akin to the Y-combinator (see page 86), a constructor for functions using operators that encapsulate the recursion or iteration concept analogous to the summation operator in orthodox mathematics. But, developing this thought I found even more reliance on variable symbols and tricks for modifying stored data. I arrived at the time and place of the talk in a highly dubious frame of mind, expecting a dirty trick.

At this talk I was introduced to S-K combinators, which, unlike pictures of Lilly, did not make my life complete, but certainly did opened my eyes to the point that such an approach was possible. The central issue is really one of how to move the values around.

If I have a number of basic functions available, and operators on those functions, then I can build up more functions. I do not need to have any variables to do this, for example, a summation operator might be defined so that

$$\texttt{(sumOp 0 f) n} = \sum_{i=0}^{n} f(i).$$

An example program is `(sumOp 0 sqr)`, and contains only constants, no variables at all.

In Haskell, we can use a function such as `sub b a = b - a` in the definition `f x = (sub 5) x`, but since `x` occurs on the end of the expression on both sides we can instead say `f = sub 5`. The `x` was an arbitrary

place holder. So far so good. But when we try to define `f(x) = x-4`, we get `f x = sub x 4`. Now `x` is in the middle and cannot be thrown away. How do we avoid using it? We include the operator, `R f x y = f y x` and define `f x = R sub 4 x`. Now, `f = R sub 4`, and we have eliminated `x`.

A moment's reflection should now convince the reader that what we need is the ability to define arbitrary expression construction where the (unknown) arguments are inserted into an expression being built. At this point it is logically possible that, there being an infinite number of expressions, we might need an infinite number of constructors. However, it has been shown that two are sufficient.

```
K x y = x
S x y z = x z (y z)
```

Defining `I = S K K` so that `I x = x` is worthwhile and we can reduce an SK expression as follows:

```
(S (K S) K) I) f x = f (f x)
(S (S (K S) K) I) f x
S (K S) K f (I f) x
(K S) f (K f) f x
S (K f) f x
(K f) x (f x)
f (f x)
```

The original thought behind SK combinators was that they would make compilation of functional languages efficient, but the size of the expression tends to double each time a variable is eliminated, thus making the expression horribly large.

However, if we abandon minimality, and use other combinators, for example, `C x y z = x (y z)`, then we can reduce the size of some expressions, `(S C I) f x = f (f x)`. The use of a much larger set of rather more complicated "super" combinators, can actually make this approach work. Bu, it is a non-trivial exercise.